THE IMPACT OF MESSAGE TRAFFIC ON MULTICOMPUTER MEMORY HIERARCHY PERFORMANCE

BY

SCOTT DOV PAKIN

B.S., Carnegie Mellon University, 1992

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois

Abstract

Multicomputer cache performance is highly sensitive to interprocessor message traffic. The widening gap between microprocessor speeds and primary memory latencies means small increases in the number of cache misses can have a severe impact on application performance. It is therefore critical to reduce cache misses. While there are a number of factors that contribute to increased cache misses, of particular concern to system designers and application writers alike is multicomputer message traffic. In this thesis, we examine the extent to which handling message traffic increases cache misses.

Multicomputer network interfaces are connected to the local memory hierarchy. The specific connection point affects performance. Connecting the network interface to higher (i.e. smaller, but faster) levels of the memory hierarchy improves a node's parallel performance by reducing communication latency. In contrast, connecting the network interface to lower (i.e. larger, but slower) levels of the memory hierarchy improves sequential performance by retaining program data in the higher levels. However, the extent to which message handling displaces program data in the cache in either scheme was has not previously been explored.

Using register-transfer-level trace-based simulators of two representative multicomputer nodes, we performed extensive tests (over 70 billion memory references) with a variety of system configurations. Our results indicate that message traffic has a tremendous impact on multicomputer node performance. A multicomputer with a primary memory–network interface and average-sized cache suffers significantly more cache misses in the presence of message traffic than in its absence. For example, a memory–network multicomputer with a 16 kilobyte (KB) cache that processes an average of one word of message data per 50 clock periods can be expected to observe up to 20% more misses on a given application than a similarly-configured uniprocessor. Furthermore, increasing the cache size increases a multicomputer's sensitivity to message traffic. Therefore, a multicomputer with a cache–network interface that performs an average amount of communication per unit time benefits less from a larger cache than would a uniprocessor. Our results show that doubling the cache size from 16 KB to 32 KB decreases cache misses by 55% for one application on a uniprocessor but by only 35% on a cache–network multicomputer. Finally, we show that for a fixed message volume, control (i.e. short) messages affect the cache more than data (i.e. long) messages. After presenting our findings, we discuss the significance of these findings and propose starting points for future research.

Acknowledgements

"I can't give you brains, but I can give you a diploma."

L. Frank Baum, *The Wizard of Oz*

To start, I would like to thank my advisor, Professor Andrew A. Chien, for his advice and guidance on my research, for his comments and criticism on my thesis and previous documents, and for "giving me brains"—helping further my education through innumerable technical discussions on a variety of topics. I'm thankful that for all the delays and pitfalls I faced along the way to finishing my research and thesis, he didn't give up on me.

Second, I am much obliged to John Plevyak and Vijay Karamcheti for reviewing this thesis and providing me with a wealth of insightful comments. Their feedback helped clarify and improve a number of sections that were evidentally not as lucid as I had thought. Also, Julian Dolby deserves thanks for frequently suggesting better phrasings for awkward-to-express sentences.

I'm grateful to everyone in the Concurrent Systems Architecture Group for some terrific arguments and discussions on everything from politics to technology to *The Simpsons*, for their wit, humor, intelligence, and good cheer, and for providing a atmosphere in which everyone helps everyone else along. I could not have hoped for a better set of peers.

Finally, I thank my parents for their bottomless love, support, and encouragement. I thank my grandparents for never stopping loving me, even though I've been so busy with research that I've had far too little time to spend with them. And I thank my sister for . . . I don't know; she's probably done something nice for me at some point.

Table of Contents

Chapter

List of Tables

List of Figures

Chapter 1

Introduction

High-performance computing is crucial to manufacturing, science, defense, and other areas. Solutions to current problems in these fields rely on accurate models of natural and/or artificial phenomena to reduce cost, increase safety, and improve quality of life. However, accurate models require tremendous computing power. Only the highest-performing computers will be capable of supporting the computational demands of applications as diverse as aerospace simulations, quantum chromodynamics calculations, and global climate change predictions. Realizing how important those applications—and therefore, high-performance computing—are to all sectors, the U.S. government recently formed the Federal High Performance Computing and Communications Program (HPCC), whose charter includes "extend[ing] U.S. technological leadership in high performance computing and computer communications" [10].

Traditionally, high-performance computing meant fast uniprocessors—computers with a single, powerful processor and a high-speed memory system. However, due to physical limitations, such as the speed of light, it has become increasingly difficult to produce faster uniprocessors. To achieve their goal of developing high-performance computers capable of sustaining over a trillion operations per seconds (teraops), the HPCC promotes scalable, parallel computers [10]. In addition to being a more promising technology in terms of performance, parallel computers have the additional advantage of comparatively low cost, as—unlike high-performance uniprocessors—they are generally built from commodity components.

A parallel computer is composed of multiple computers (*nodes*) that cooperatively solve problems by working on separate portions of the computation and communicating status information and data with other nodes. Each node, like an entire uniprocessor system, contains a processor and memory. The parallel computers we consider in this thesis are *scalable*, which means that aggregate computational power available to an application can be increased by increasing the number of nodes. Parallel computers are generally converging to multiple instruction/multiple data (MIMD) [18] architectures. Each processor in a MIMD machine operates independently on a subcomputation and synchronizes with other processors as needed. In this thesis, we focus on one subcategory of MIMD machines, called *multicomputers*. Multicomputers are distinguished by their distributed memory and message-passing style of communication. That is, communication is performed by passing data in the form of a *message* from one node's memory to another's.

1.1 The Problem

One important concern in multicomputer node architectures not shared by uniprocessor architectures is the relation of communication, computation, and memory performance. Computer systems generally contain a small amount of fast memory and a large amount of slower memory. Both communication and computation benefit from using the fast memory. Communication benefits from fast memory because messages are sent between nodes' memory. Computation benefits from fast memory because program data is stored in memory. However, because the quantity of fast memory is limited, optimizing an architecture for communication performance (by storing message data in the fast memory) implies that computation performance suffers because its data is relegated to slower memory. Similarly, optimizing an architecture for computation performance decreases communication performance. In this thesis, we show the impact of message traffic on memory performance for two architectures and show how that impact varies across a range of scenarios.

1.1.1 Memory Hierarchies

An ideal memory system would be infinitely large and infinitely fast. But due to physical limitations, the ideal is unachievable and hence, tradeoffs must be made in system design. Faster memories are expensive and thus, necessarily small. Slower memories are comparatively inexpensive, and can therefore be larger. But rather than selecting a single memory size and speed, system designers construct memory systems out of small amounts of fast memory, and a large amounts of slow memory. The memory is organized hierarchically, with the smallest, fastest memory kept closest to the central processing unit (CPU), and the largest, slowest memory farthest away. The largest, slowest, but least expensive memory—*primary memory*—is at the bottom of the memory hierarchy[1], with levels of increasingly smaller, faster, but more expensive, memory—generally organized as a *cache*—higher up. Cache memories [32] contain a dynamically-changing subset of lower memories in the hierarchy. When the CPU makes a request (a `LOAD` or `STORE`) to a memory address, the first level of cache memory is searched. If the data is present in the cache, the cache returns or modifies the data (as appropriate). This is called a *cache hit*. If the cache does not contain the address (a *cache miss*), the memory request is propagated down the memory hierarchy until the address is found. The goal of a memory hierarchy is to create the illusion of a memory that is both large *and* fast. The success of the illusion depends on the likelihood that data the CPU needs is located in the fast cache memory.

Figure 1.1 shows a sample memory hierarchy. To put the relative speeds and sizes of each level in perspective, we show, the memory hierarchy figures of the DEC 4000 AXP series of workstations. Table 1.1 lists the size of each level of the memory hierarchy, measured in kilobytes (KB), and the corresponding latency, measured in nanoseconds (ns). Data used in the table is based on that in [14, 27] (160 megahertz (MHz) DECchip 21064 CPU, maximum memory/cache sizes, bus latency included). Note that each level in the memory hierarchy is approximately two orders of magnitude larger than the immediately higher level and one order of magnitude slower.

[1] In some systems, the memory hierarchy is extended to secondary and even ternary memory. However, in this thesis, we will assume that primary memory is the lowest level of the memory hierarchy.

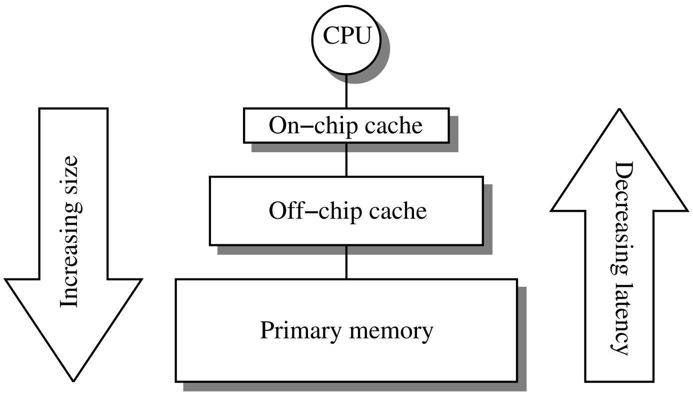


Figure 1.1: Sample memory hierarchy

Component	Size (KB)	Latency (ns)
On-chip cache	16	6.25
Off-chip cache	4,096	25
Primary memory	2,097,152	275

Table 1.1: Memory hierarchy of the DEC 4000 AXP

1.1.2 Network Interfaces

Multicomputer communication involves transmitting data from one node to another over a high-speed interconnection network. The network interface provides the gateway between the memory hierarchy and the network, and logically connects to one or more levels of the memory hierarchy (Figure 1.2). The CPU is portrayed as a memory–network connection point because the CPU register file, sometimes considered a "compiler-controlled cache," forms the top of the memory hierarchy.

Communication performance is largely a function of memory performance. Communication performance is measured in terms of *latency* and *bandwidth*. Latency is the time it takes for the first piece of data to arrive at the destination node, and bandwidth is the message volume transmitted per unit time. Because memory is the source and destination of communicated data, communication latency can be no lower than memory latency (a few hundred nanoseconds), and communication bandwidth can be no higher than memory bandwidth (a few hundred megabytes/second).

1.1.3 The Conflict

Computation performance, like communication performance, relies on rapid access to data. However, the limited size of the upper levels of the memory hierarchy is insufficient to store both communication data (i.e. incoming and outgoing messages) and computation data (i.e. application data structures), let alone either of them in its entirety. Nevertheless, the more upper-level memory is available to communication or computation, the faster its operations will execute. Therefore, to make room for message data, other data (such as that in use by

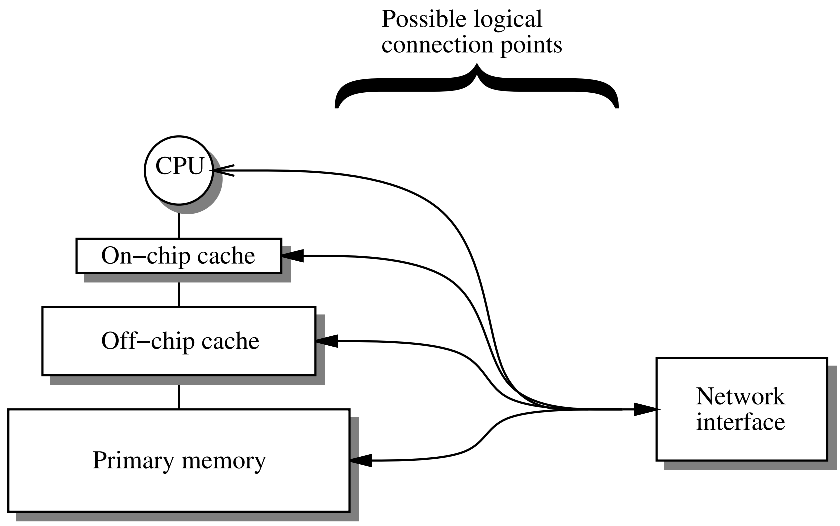


Figure 1.2: Logically connections between the network interface and the memory hierarchy

an application) must be displaced from its position in the memory hierarchy to a lower level. Similarly, when the CPU pulls application data up the memory hierarchy, it pushes other data (such as that in use by messages) down the memory hierarchy. For one piece of data to become more readily-accessible, another must become less readily-accessible.

As a result of communication and computation performance being tied to memory performance, the point at which the network interface and memory hierarchy logically connect is important. Logically connecting the network interface to one of the higher levels of the memory hierarchy enables lower-latency communication, but displaces program data, making it less readily-accessible. In contrast, logically connecting the network interface to one of the lower levels of the memory hierarchy preserves important data in fast memory, but increases communication latency. The tradeoff is thus that either message data or program data can be accessed with low latency.

Consider two alternatives: logically connecting the network interface to primary memory or logically connecting the network interface to the cache (assuming a single cache in this model). In the former (Figure 1.3), the network interface writes communication data directly to primary memory. While primary memory latency degrades communication performance, cached data is not displaced. Furthermore, the CPU can continue accessing the cache while communication data is written to primary memory, thereby hiding some of the communication latency. In systems that logically connect the network interface to the cache (Figure 1.4), the CPU actively reads communication data from the network interface and writes it into the memory hierarchy. (We are assuming that the network interface cannot directly access the cache, as is generally the case for on-chip caches.) Because the cache has limited capacity, a previously-cached piece of data is displaced to make room for it. However, because cache memory is faster than primary memory, the CPU can access the communication data faster than in the previous scheme.

In practice, the interrelation between communication and computation performance is more complex than a simple "either-or." Because a message generally contain data produced by a computation, and computation generally acts on data received from a message, there is some degree of data sharing between communication and computation. If a message arriving at a

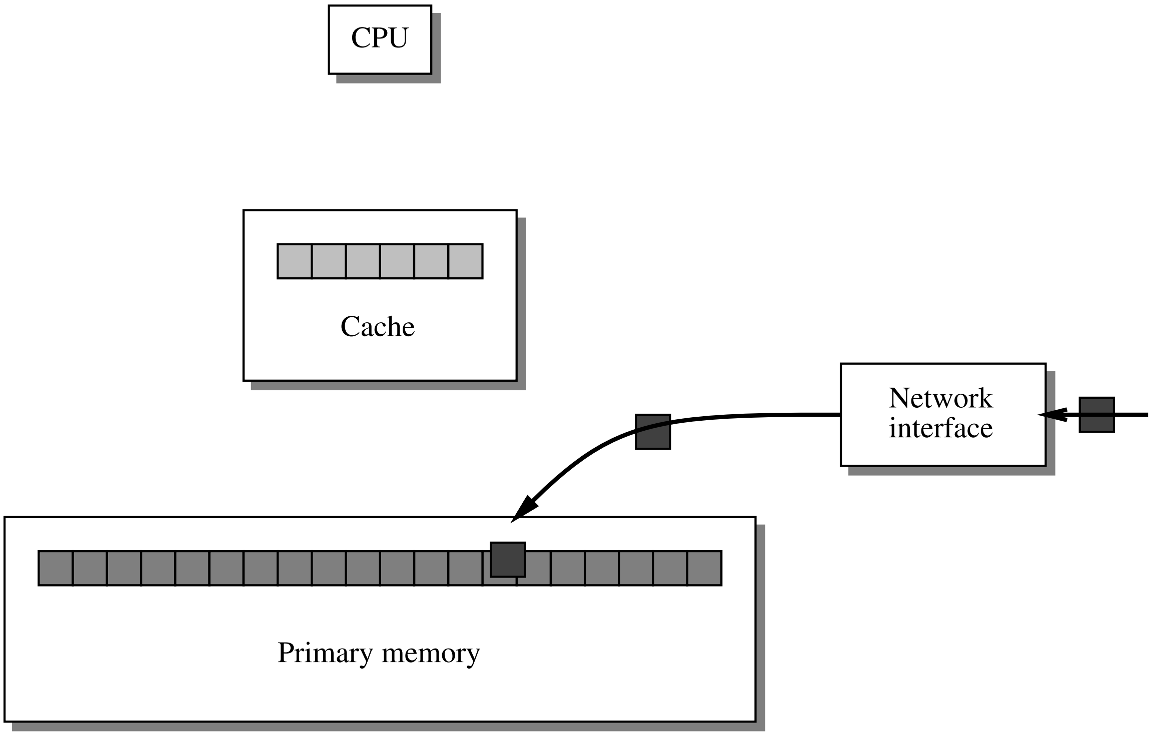


Figure 1.3: Receiving a message into primary memory

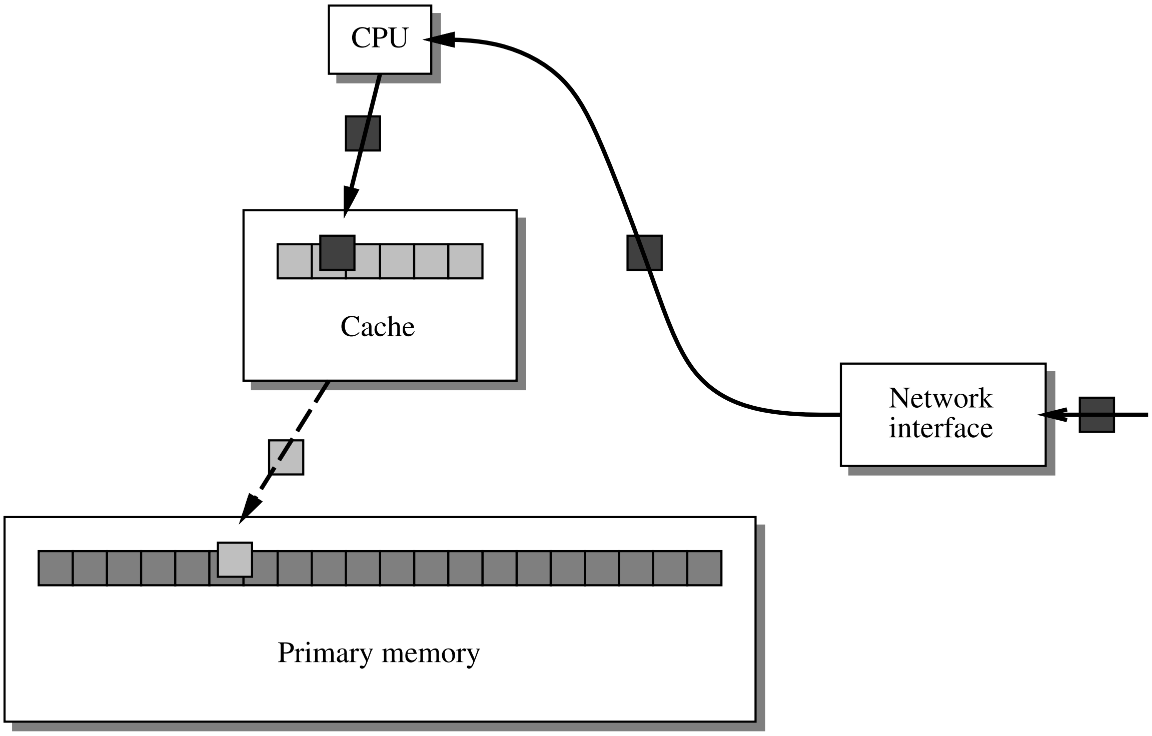


Figure 1.4: Receiving a message into the cache

node is quickly incorporated into computation, a cache–network interface is the better design because of the speed at which computation can integrate the incoming data. In contrast, if an application cannot promptly use incoming data—either the application is not ready to process message data, or only a portion of the message can be used immediately—a primary memory–network interface is the better design because it does not disrupt computation by requiring it to handle data it does not yet need. In general, the sooner message data is to be shared, the higher in the memory hierarchy it should be placed.

When the type of data sharing is not known *a priori*, system designers must decide which is more important: rapid access to program data or rapid access to message data. To determine whether network–cache or network–primary memory has superior overall performance, one must know how much message handling affects memory hierarchy performance in a variety of scenarios. That is exactly the problem we address in this thesis—quantifying message handling's impact on the cache for network–cache and network–primary memory architectures in various situations and under different system configurations.

1.2 The Approach

We constructed register-transfer-level trace-based simulators to simulate the operation of two systems that are alike in every way except for the connection point of the network interface and the memory hierarchy. One simulated system connects[2] the network interface and the cache, and the other connects the network interface and primary memory. In addition, we simulated a uniprocessor that is identical to the two multicomputers except that it lacks a network interface and therefore, does not engage in message handling. Using those three simulators, we studied message handling's impact on the cache under a variety of message rates, cache sizes, and message types.

1.3 Related Work

While multi*processor* cache performance has been studied extensively, there have been virtually no studies of multi*computer* cache performance. The primary exception is [33], in which Stunkel and Fuchs characterize the cache performance of an Intel iPSC/2 hypercube [22]. The iPSC/2 is a multicomputer composed of up to 128 Intel 80386 microprocessors, each equipped with a 64 KB off-chip cache and up to 16 megabytes (MB) of primary memory. The network interface connects to primary memory.

To perform their analysis, Stunkel and Fuchs performed trace-based simulations of an iPSC/2 running a set of parallel applications and measured the cache miss rate. Stunkel and Fuchs varied the cache size and number of processors, and performed their tests both with and without an operating system.[3] They then report the overall cache miss rate—the sum of all the processors' cache misses divided by the sum of all the processors' memory references—for each scenario.

According to [33], given a constant problem size, the cache miss rate decreases with the number of processors until the application's working set fits in the cache, and increases from there on. In addition, Stunkel and Fuchs found that applications that send and receive many

[2] In this thesis, "connects" implies a *logical* connection rather than a *physical* one.

[3] The iPSC/2 normally runs the UNIX operating system on every node.

short messages observe worse cache performance than applications that send and receive long messages in bursts with little or no intervening communication. This latter result agrees with our findings, described later in this thesis.

1.4 Results

Our results indicate that message traffic has a tremendous impact on multicomputer node performance. Among the things we found are:

- Message traffic greatly increases the number of cache misses observed by a multicomputer with a memory–network interface and average-sized cache. With a 16 KB cache, a multicomputer that processes one word of message data per 50 clock periods can be expected to observe up to 20% more cache misses than a similarly-configured uniprocessor.
- Larger caches benefit multicomputer nodes less than uniprocessor nodes. For example, doubling the cache size from 16 KB to 32 KB decreases cache misses by 55% for one application on a uniprocessor but by only 35% on a multicomputer with a cache–network interface.
- Control and data messages (i.e. short and long messages) affect the cache differently. For a constant volume of messages, short messages displace more cached data than long messages for both cache–network and primary memory–network interfaces. In addition, for some applications, cache–network interfaces displace less cached data than primary memory–network interfaces when handling short messages.

1.5 Thesis Overview

The remainder of this thesis is organized as follows. In Chapter 2, we describe a number of current research and commercial network interfaces and compare them to the basic CM-5 and Paragon designs we use for our simulations. Chapter 3 covers our experimental framework, introducing our simulators at both a conceptual and detailed level, describing the application traces we used, and presenting our simulation parameters and dependent variables. The results of our simulations are presented and analyzed in Chapter 4. There, we show the impact of message traffic on cache performance for two network interfaces and show how that impact varies with message type and cache size. In Chapter 5, we discuss our findings qualitatively, explaining the ramifications of our results on present multicomputers, and forecasting the cache performance of future multicomputers based on our results and on current trends in system design. Drawing upon those trends and forecasts, we provide suggestions for future multicomputers and applications. In Chapter 6, we summarize our results and provide some conclusions. Finally, in Chapter 7, we mention topics for future research that expand upon our findings.

Chapter 2

Background

A wide variety of parallel machines with a wide variety of network interfaces have been built. However, they illustrate only two basic choices. Section 2.1 describes systems in which the network interface connects to the cache, viz. the CM-5, T9000, AP1000, and Alewife. We note how each system addresses the problems created by that organization: cache pollution and increased CPU load. Section 2.2 describes systems in which the network interface logically connects to primary memory. While that organization does not suffer from cache pollution and increased CPU load from message handling, it introduces a new concern: increased messaging latency. We note how each of the CS-2, Paragon XP/S, EDS, and Mosaic C attempt to support fine-grained communication in view of that latency.

2.1 Cache–Network Interface

The advantage of connecting the network interface to the cache is that the speed of message sending and receiving is not dominated by primary memory latency. Message data can be accessed after only a (comparatively small) cache latency. Lower-latency communication enables finer-grained applications.

To illustrate the importance of low-latency communication for fine-grained applications, consider an application composed of seven tasks of unit time, the middle five of which are independent. In Figure 2.1(a), the seven tasks are executed sequentially, which takes seven time steps. In Figure 2.1(b), the first task communicates data to the middle five tasks, which execute in parallel and then combine their results for the final task to process. Because communication time is small, the parallel execution finishes faster than the sequential execution, taking only five time steps. Figure 2.1(c) depicts the seven tasks executing in the same manner as in Figure 2.1(b), although on a system where message handling incurs a high latency. Because a proportionally larger amount of time is spent communicating, the parallel execution of the application takes nine time steps, making it slower even than the sequential execution. Hence, if an application communicates frequently relative to the amount of intervening computation, low-latency communication is imperative for good parallel performance.

The disadvantage of logically connecting the network interface to the cache is that message handling causes cache pollution. When messages are constructed in or received into the cache, they displace program data previously resident. If the displaced data is no longer needed, then there is no penalty for message handling. If, however, the displaced data is needed again, the

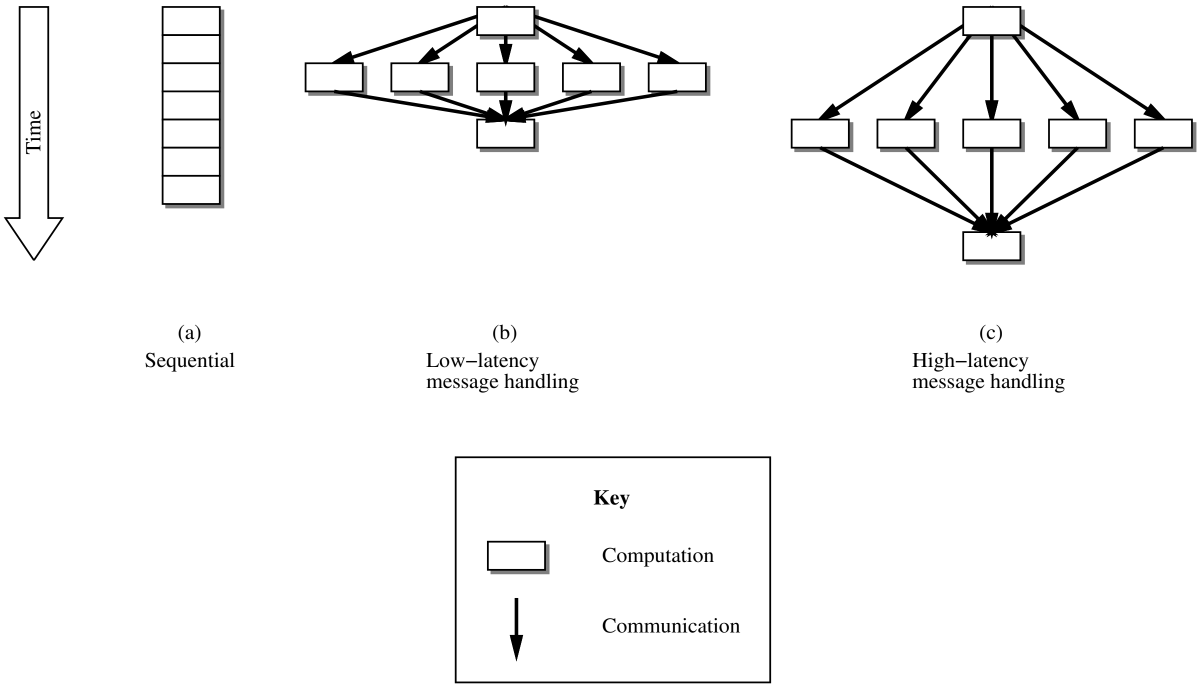


Figure 2.1: Communication granularity

CPU must incur a primary memory latency to access it. In the absence of message handling, it need incur only a cache latency.

2.1.1 Basic Design

The basic model used by systems with cache–network interfaces is that the CPU connects directly to a cache, and the cache, primary memory, and network interface share a bus (although the network interface communicates only with the cache). The network interface communicates only with the CPU, which must initiate the communication. In that sense, the network interface behaves like primary memory: It passively awaits the CPU's `LOAD` and `STORE` commands, which it then responds to by returning or accepting data. Two examples of the basic cache–network interface model are the CM-5 and the T9000.

One of the design goals [29] of the CM-5 was to reap the benefits of a MIMD architecture, while still maintaining compatibility with the CM-5's single instruction, multiple data (SIMD) [18] predecessor, the CM-2. Because SIMD architectures communicate frequently, providing low-latency communication in the CM-5 was crucial for CM-2 compatibility. Therefore, the CM-5's network interface was designed to logically connect to the cache, as depicted in Figure 2.2. The network interface lies between the cache and memory, yet logically connects only to the cache.

By optimizing for low-latency communication, the CM-5's network interface supports the CM-2's fine-grained SIMD model. However, without access to primary memory, the network interface is unable to send or receive memory- (but not cache-) resident data directly. That can be a problem when an application needs to send large quantities of data from memory. In that situation, memory data must be loaded into the cache prior to transmission, which can cause

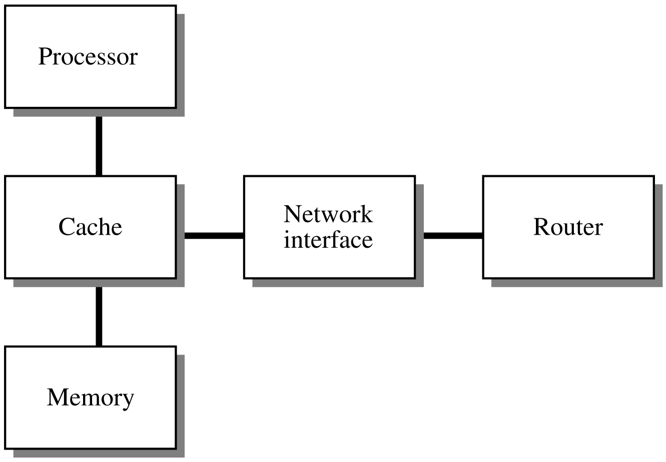


Figure 2.2: CM-5 block diagram

cache pollution. Consider a node that needs to transmit a large array from primary memory. Loading each element of the array into the cache can displace a significant quantity of other program data.

An additional drawback of the CM-5's node architecture is that the CPU is involved in message transfers. The CPU must explicitly move message data from the cache to the network interface. Thus, the CPU must postpone computation during message handling.

The T9000 Transputer [28] is designed specifically for fine-grained MIMD parallelism. It implements a custom-designed CPU, network interface, and cache[1] on a single chip, reducing latency but increasing design complexity.

As Figure 2.3 shows, the processor, cache, and network interface are interconnected via a crossbar instead of a (slower) bus. The network interface uses a direct memory access (DMA) controller to transport data between the network and the cache. Unlike the CM-5, the T9000 frees the CPU from message handling. The CPU can continue computing during message transfers, as long as it does not require data from the cache bank in use by the network interface. (The cache is divided into four banks.) This increases computational throughput relative to the CM-5.

2.1.2 Cache- and Memory-Network Interface

A variant on the model described in Section 2.1.1 is to logically connect the network interface to both the cache and primary memory. The advantage of a dual connection is that long messages can be transmitted using primary memory to minimize cache pollution, and short messages can be transmitted using the cache to minimize latency. Two examples of architectures using a cache- and memory-network interface are the AP1000 and Alewife.

The AP1000 computer shares with the CM-5 the goal of low-latency communication [24]. It, too, logically connects the network interface and the cache, but in a substantially different manner from the CM-5 (Figure 2.4). Primarily, the AP1000 logically connects the network interface to both the cache *and* primary memory. This organization alleviates cache pollution

[1] The T9000 can be software-configured to treat half or all of its on-chip cache as primary memory.

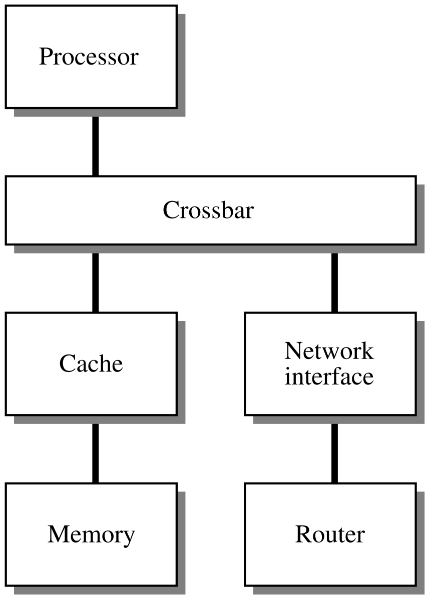


Figure 2.3: T9000 block diagram

for large, primary memory transfers, yet does not preclude low-latency communication using short, cache-based transfers. The application can choose on a per-message basis an appropriate level of the memory hierarchy to use for communication.

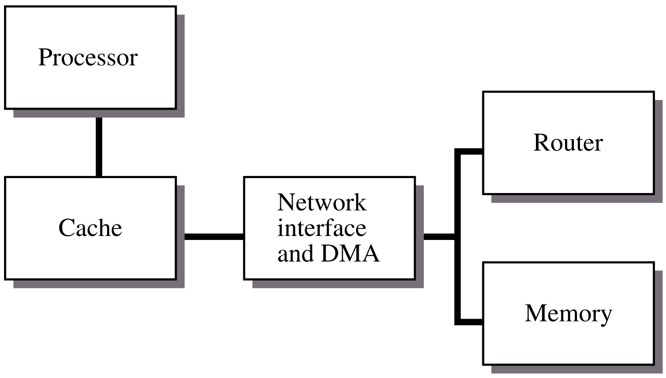


Figure 2.4: AP1000 block diagram

While the CM-5's CPU must transport data between the network interface and the cache, the AP1000's network interface can access the cache directly. That architectural design addresses the problem of occupying the CPU for message sends by enabling the CPU to continue computing while the network interface transports data between the cache and the network. However, it introduces a new architectural concern: for the network interface to be able to access data directly from the cache, either the cache must be off-chip, or the microprocessor must allow external devices access to the on-chip cache. The former is increasingly difficult given increasing CPU clock rates, and the latter requires functionality not supported until recently by current-day microprocessors (in the form of cache-coherence protocols, which can provide limited forms of access to the on-chip cache).

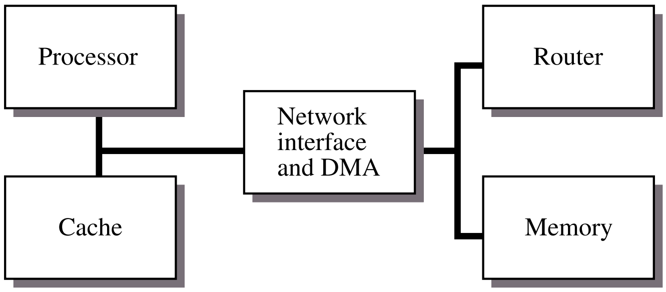


Figure 2.5: Alewife block diagram

The Alewife machine [25] shares the AP1000's high-level architectural design. Both systems give the network interface access to both the cache and primary memory. However, as Figure 2.5 indicates, the Alewife's network interface is additionally directly accessible by the CPU. There are thus three ways to send messages in the Alewife:

1. DMA transfers, from both the cache and primary memory. (That is, data are sent from the highest level of the memory hierarchy at which they are located.)

2. Writing directly to network interface queues

3. Cache-coherent memory operations

The last of those provides particularly low-latency communication. In the Alewife, nodes can cache data from remote nodes' primary memories, and if the data is changed, all caches containing that data are notified. Hence, a single `STORE` instruction automatically sends messages to each node that has the corresponding data cached. Similarly, a single `LOAD` instruction automatically sends a message to the corresponding data's "home" node, which sends back a message containing the data. In contrast, message passing on the other systems described in this section requires message characteristics (i.e. destination node, number of bytes to transfer, and starting address) to be communicated to the network interface before the message is launched. The Alewife's ability to perform implicit message passing reduces software overhead and enables finer-grained communication than would otherwise be supported. The drawback of implicit message passing is that messages may be sent even if doing so is unnecessary. That is, as long as data is cached in multiple nodes, any modification to that data will result in message sends, even if the data is needed only by the node that modified it. This causes unwanted message traffic, which detracts from the network bandwidth shared by all nodes.

2.2 Memory-Network Interface

To reduce cache pollution, systems can logically connect the network interface to primary memory. In this type of system, the network interface is equipped with a DMA controller, which the CPU programs to send and receive data. Once the CPU sets up the DMA controller, the CPU can continue computing, even performing cache operations. Overlapping computation and communication allows applications to make progress during message handling, as long as they do not require access to incoming message data.

The disadvantage of logically connecting the network interface to primary memory is an increase in messaging latency. Because primary memory must be accessed for all message transmissions, communication latency can be no less than primary memory latency, which can be fairly large—on the order of 50–70 ns, incurred for each memory access. As a result, applications running on systems that logically connect the network interface exclusively to primary memory must be an order of magnitude more coarse-grained to observe an advantage from parallelism on a node with a primary memory–network interface than on one with a cache–network interface.

2.2.1 Basic Design

The basic model used by systems with memory–network interfaces is similar in structure to that used by systems with cache–network interfaces. In both, the CPU connects directly to a cache, and the cache, primary memory, and network interface share a bus. The difference is that the network interface contains a DMA controller and can therefore access memory autonomously from the CPU. In that sense, the network interface behaves like a CPU itself: It actively sends `LOAD` and `STORE` requests to primary memory. An example of the basic memory–network interface model is the CS-2.

The CS-2 [21] logically connects the network interface to primary memory in an extremely straightforward manner. As Figure 2.6 indicates, the network interface and DMA controller lie on the bus connecting the processor and memory, and the router connects to the network interface.

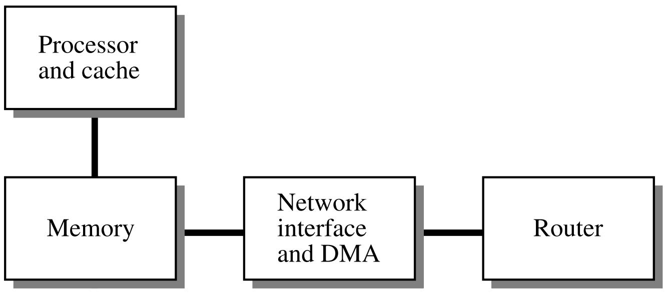


Figure 2.6: CS-2 block diagram

Although the basic memory–network interface is intended for coarse-grained parallelism (due to the high communication latency incurred by the need to access primary memory), the CS-2's network interface deviates from the basic design by implementing a few operation-specific fine-grained features. For example, it supports atomic swap, increment, and test-and-store synchronization primitives on remote synchronization variables, and it can poll semaphores located on a remote node. (That is, it determines if a local and remote variable are $=, \neq, <, >$, $\leq$, or $\geq$.) While those operations are certainly fine-grained—they represent small amounts of computation between communications—shifting the responsibility for executing them from the CPU to the network interface enables the remote node to compute while its network interface handles the fine-grained operations. Thus, rather than reducing latency, the CS-2 masks it by overlapping fine-grained operations with coarse-grained computation.

2.2.2 Messaging Coprocessors

Logically connecting the network interface to primary memory does not imply that the cache is immune to message handling effects. Operating system or run-time system involvement in message construction, packetization, and protocol processing affects the cache. One way to alleviate this problem is to add a messaging processor. Two examples of systems that follow this approach are the Paragon XP/S and the EDS.

The Paragon XP/S [23] system tries to minimize cache pollution by adding a second processor (with a separate on-chip cache) specifically to handle messaging operations (Figure 2.7). Thus, the "compute" processor's cache is largely unaffected by communication; only the "messaging" processor's cache suffers.

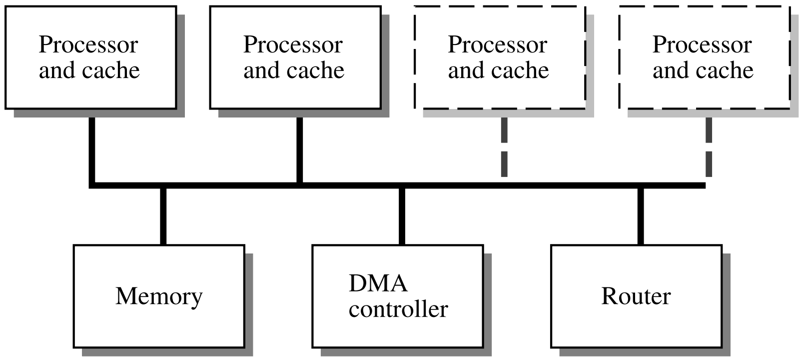


Figure 2.7: Paragon block diagram

While the Paragon reduces cache misses by adding an additional processor with on-chip cache, an alternative way to reduce cache misses is merely to add an off-chip cache. However, adding a second processor additionally reduces costly context switches from the user's process to the operating/run-time system and back for each communication. With two processors, the compute processor can stay in user mode and the messaging processor can stay in the part of the operating/run-time system that handles messages, and neither needs to context switch to communicate. The disadvantage of dedicating one processor to computation and one to communication is that computation-intensive tasks are limited to using half of the available CPU power.

The European Declarative System (EDS), a multicomputer produced by a consortium of Bull, ECRC, ICL, and Siemens [34], is architecturally similar to the Paragon, although it addresses the dedicated-processor issue. The main differences between the EDS and the Paragon are:

1. The EDS does not dedicate its messaging processor to messaging; it is also allowed to compute.

2. The EDS provides a lower-latency connection from the messaging processor to the network interface.

The advantage of allowing the messaging processor to compute is that it enables up to twice the computational throughput of a system (like the Paragon) that restricts the messaging processor to message handling. The disadvantage of performing computation on the messaging processor is that doing so forfeits the benefits of a reduced number of context switches and a lower cache miss rate for program data in the presence of message handling.

While the Paragon distinguishes the compute and messaging processors through software, the EDS distinguishes them through hardware. Only the EDS' messaging processor has low-latency access to the network interface. The network interface is connected between the CPU and the off-chip cache (Figure 2.8) and can therefore accept commands directly from (and return status information directly to) the CPU. While this feature does not reduce communication latency (which is still dominated by primary memory latency), it does reduce the amount of time the CPU is involved in message transfers.

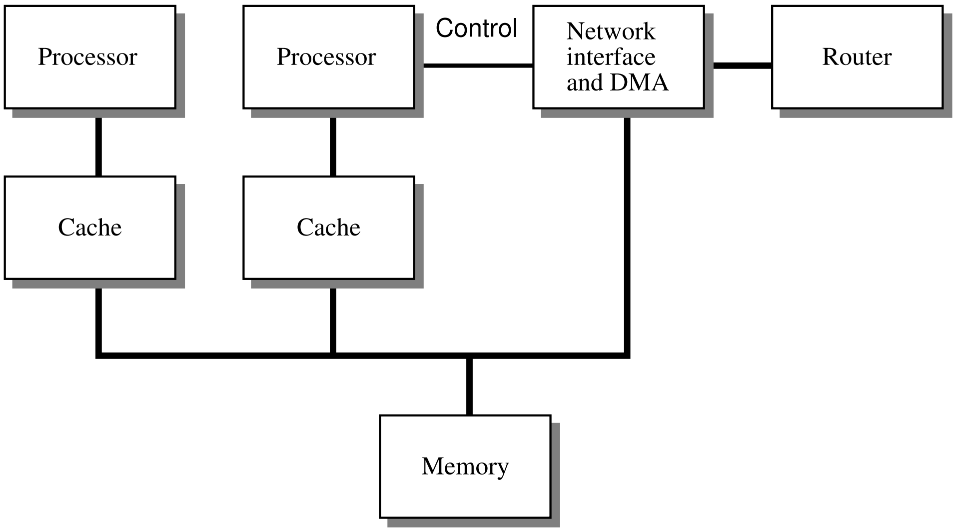


Figure 2.8: EDS block diagram

2.2.3 Fine-Grained Systems

Memory–network interfaces do not preclude fine-grained communication. However, they do require different *implementations* from those described in Sections 2.2.1–2.2.2. Specifically, memory–network interfaces require a higher level of system integration to prevent primary memory from becoming a communication bottleneck. The Mosaic C is an example of a fine-grained multicomputer.

Like the CS-2, Paragon, and EDS, the Mosaic C [30] logically connects the network interface to primary memory. However, unlike those other machines, the Mosaic C is designed for fine-grained parallel applications. Fine-grained applications require low-latency communication, making the cache in the Mosaic C's block diagram (Figure 2.9) conspicuously absent. The Mosaic C does not contain a cache because primary memory is not a bottleneck for the node's comparatively slow CPU. While the slow CPU speed implies multiple Mosaic C nodes must be used to achieve the same throughput as a single i860/XP or SPARC microprocessor, the Mosaic C is intended to be used for very fine-grained applications running on a large number of nodes. In that scenario, the ability to exploit parallelism is more important to performance than single-node computing power.

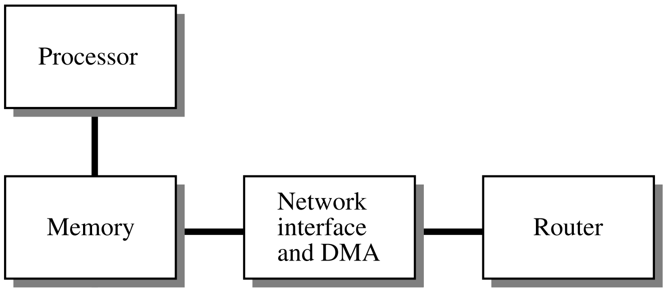


Figure 2.9: Mosaic C block diagram

The Mosaic C node is only a single chip. The advantage of a single-chip design is that it reduces inter-component latency by reducing the frequency data must go off-chip. In the Mosaic C, only interprocessor communication must cross chip boundaries—not cache/memory operations, as is the case for the other systems described in this section. Two disadvantages of single-chip designs are limited chip area and high design cost. Because chip area is limited, fitting the CPU, network interface, router, and primary memory on a single chip implies that each component is generally less powerful than it would have been, given the entire chip.

Like the Mosaic C, the J-Machine [12] is a single-chip fine-grained machine. What distinguishes the J-Machine from a memory hierarchy/network standpoint is that while messages can be sent and received using primary memory, the can additionally be *sent* directly from CPU registers (Figure 2.10). Message reception is handled without any CPU involvement; arriving messages are automatically placed in a circular queue, where they are read by the CPU. The motivation for this structure is that in fine-grained applications, data intended for transmission is likely to be in the topmost levels of the memory hierarchy, and should therefore be sent from there. Arriving messages, however, initially go to lower levels of the memory hierarchy so as not to disturb the computation in progress. Although their architectures are different in regard to their interface between the network and the memory hierarchy, the J-Machine and the Mosaic C share their advantages (low-latency communication) and disadvantages (limited chip area).

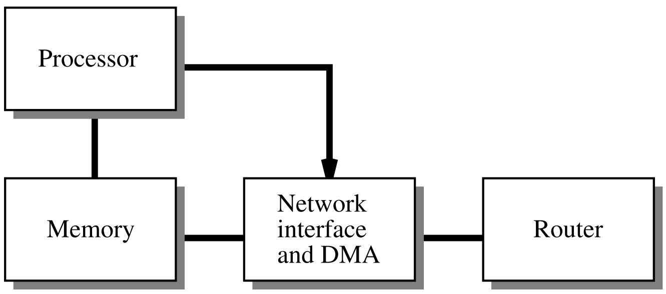


Figure 2.10: J-Machine block diagram

2.3 Summary

In this chapter, we examined a number of existing multicomputer nodes, paying particular attention to the way each interfaces the network and the memory hierarchy. While these machines exhibit a variety of features, each can be placed in one of two main categories: cache–network interfaces and memory–network interfaces. Common to systems employing the former architecture is the CPU's ability to explicitly manage data movement between the network interface and the cache. And common to systems employing the latter architecture is the CPU's dissociation with communication; the CPU merely sets up a DMA transfer and lets the DMA controller take over.

There are benefits to both schemes. The basic cache–network interface supports fast access to message data, but requires design modifications to minimize cache pollution produced by message data that is not immediately usable. For example, the AP1000 and Alewife address that problem by supporting DMA-based message transfers in addition to CPU-based ones. The basic primary memory–network interface decouples communication from computation but requires design modifications to support fine-grained communication. For example, the CS-2 provides special support for specific synchronization operations that run with low overhead. In this thesis, we examine the two basic models in an effort to quantify how well each performs in a variety of circumstances.

Chapter 3

Experimental Framework

System designers need to know what memory hierarchy–network interface has superior performance in what situations in order to optimize architectures for their particular design goals. Furthermore, application writers could use information about each node architecture to optimize their applications and to better explain performance data. For example, if logically connecting the network interface to the cache results in a vastly increased number of cache misses, that could explain worse-than-expected application performance.

Unfortunately, analytic modelling of the memory–network interface is infeasible because of the complexity of memory and network access patterns. In analytic models, complexity widens the inherent gap between insight and accuracy. Simple models of memory and network access patterns are not accurate enough, while complex models may be more precise, yet are difficult to interpret. The complexity of the interaction between communication and computation eliminates the middle ground (models that are moderately accurate and provide moderate insight).

At the other extreme from analytic modelling is comparing actual hardware implementations of the different interfaces between the memory hierarchy and the network. That is, one could build two systems—one that logically connects the network interface to the cache and one that logically connects the network interface to primary memory—and compare execution times on the actual machines. The problem with that approach is the high cost and design time involved in producing and testing new systems. Comparing existing systems that use different network interfaces is not a solution because there are far too many variables, including different processors, clock speeds, network topologies, chip technologies, and run-time systems. It is therefore difficult to make a fair comparison.

We pursue an experimental approach to evaluating logical connection points of the network interface to the memory hierarchy. The approach we use in this thesis lies between analytic models and performance measurements of hardware implementations in terms of insight and accuracy. We use simulators to compare systems that are identical except for the way they handle interprocessor communication. A *simulator* is a program that models the behavior of a system. That is, given the same inputs and simulation parameters, a simulator and a hardware implementation should ideally produce identical results. For the purpose of comparing architectures, simulators have the following advantages over analytic models and performance measurements of hardware implementations:

- Systems can be compared fairly and accurately.

- Results are repeatable.

- Parameters can be precisely controlled.

The simulators we created for this study are specified with a fine degree of detail and exhibit the features listed above.

In this chapter, we describe the conceptual model behind our simulations in Section 3.1, followed by a structural description in Section 3.2. In Section 3.3, we describe the traced applications. And, finally, we describe the simulation parameters and dependent variables in Sections 3.4 and 3.5, respectively.

3.1 Conceptual Model

To evaluate network interface performance, we simulated three machines:

- A multicomputer with a cache–network interface,
- A multicomputer with a primary memory–network interface, and
- A uniprocessor, which has no network component.

The uniprocessor serves as a performance baseline in our study.

Figure 3.1 illustrates our simulation model.[1] Our simulators are *trace-based*, which means they take as input the memory references made by an application. A *trace generator* monitors the application's execution and records the addresses referred to in all dynamically-issued `LOAD` and `STORE` instructions. (Note that this process is independent of the cache.) Because this study focuses on the memory hierarchy, only memory instructions are traced. The address trace and a set of simulation parameters are then used as input to a machine simulator. The simulator then passes each address through the memory hierarchy as if it were an actual machine that had issued a `LOAD` or `STORE` instruction.

3.2 Simulator

3.2.1 Overview

Because the goal is to compare node architectures, not specific machines, we separated *architecture* and *implementation*. To do so, we constructed three simulators (cache–network, primary memory–network, and uniprocessor) that are identical except for the manner in which they interface the network to the memory hierarchy. Nodes are simulated at the register-transfer level, which means that we modelled the *functionality* of the node's components without modelling their exact *implementation*.

3.2.2 Simulation Tools

The simulators were written using an event-driven simulation engine called Parsim [4] and a simulation language called CARL [3]. CARL—Computer Architecture Research Language—is a C-like language[2] augmented with support for the Parsim simulation model.

[1] **Definition:** simulation $\equiv$ simulator + inputs

[2] In fact, the CARL compiler generates C code with function calls to the Parsim library.

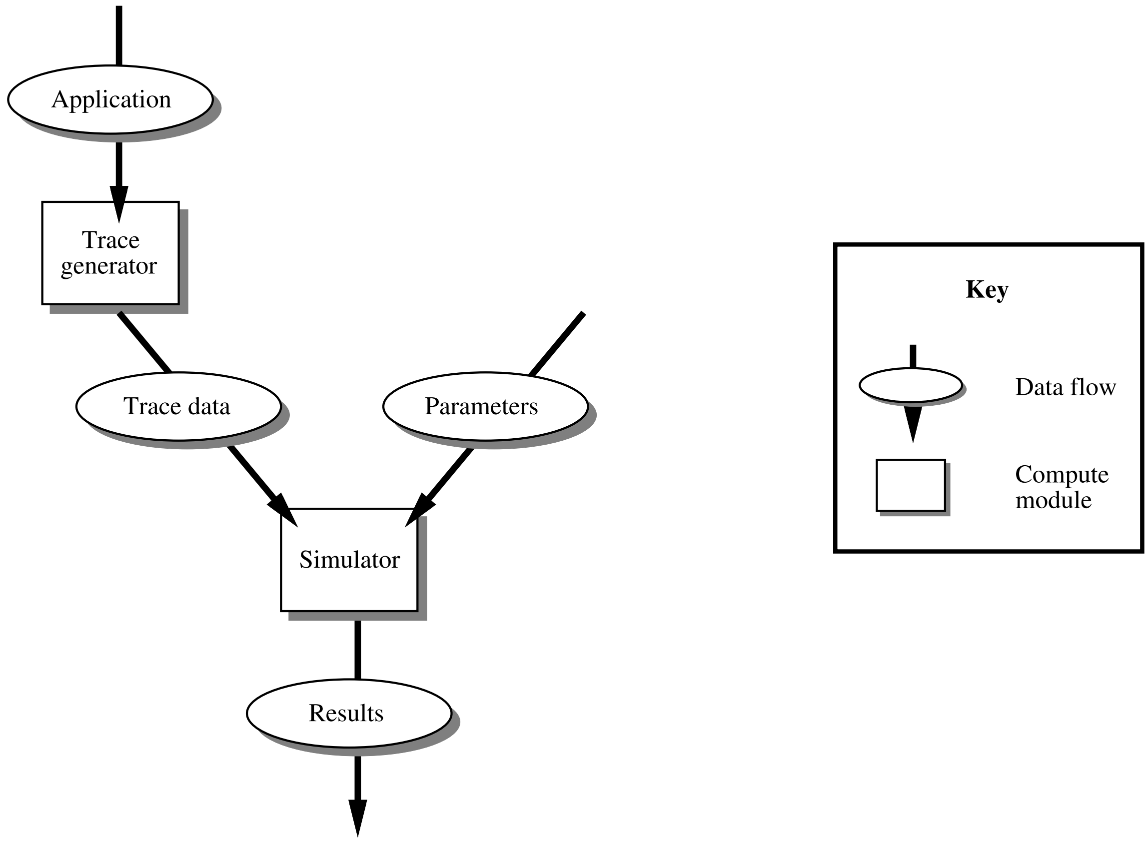


Figure 3.1: A trace-based simulation

The simulation model Parsim uses is that of a set of interconnected functional modules, or *components*. Each component defines some number of input and output wires ("nets") and contains internal state. Nets are typed and can contain:

- A data value of type `char`, `short`, `long`, `float`, or `double`,
- High impedance,
- Unknown, or
- Error

In addition, there is a special valueless net type called `clock`, which exists to respond to clock events (described below). Components can (and often do) contain other modules as subcomponents.

Changes in net contents in Parsim are considered *events* and occur at a specific time (measured in *ticks*, an arbitrary unit of time) on a global clock. Parsim's main loop is thus:

while the event queue is not empty **do**
    Skip ahead to the earliest time for which an event has been posted.
    Concurrently perform all events scheduled at that time.
end

```
COMPTYPE Register32
INPUTS
    clock clk : assign_value;       /* Invoke assign_value on clock event */
    long input;                     /* 32-bit input to register */
OUTPUTS
    long output;                    /* 32-bit output of register */
ACTION assign_value
    output = input after 5;         /* Output <-- input after 5 Parsim ticks */
    enable up clk;                  /* Activate clk on next up clock edge */
INIT
    output = 0 after 0;             /* Initialize output to zero immediately */
    enable up clk;                  /* clk is activated on first up clock edge */
BEGIN
END
```

Figure 3.2: CARL code for a 32-bit register

When one of a component's inputs changes, the module is notified. It then performs some function (an *action*), which optionally posts "change output net" events for some future time.

An example of CARL code—a module describing a 32-bit register with a propagation delay of 5 ticks—is shown in Figure 3.2. The Register32 component takes two inputs (a `long` and a `clock`) and produces one output (a `long`). When the "clock up edge" event is posted (automatically, by the Parsim engine), `clk` responds by invoking the `assign_value` action. `assign_value`, in turn, posts an event to assign the `input` input net to the `output` output net after a delay of 5 ticks on Parsim's clock and specifies it wants to be invoked again on the next "clock up edge" event. Note that our simulators' components are significantly more complex than what is shown in Figure 3.2 and tend not to be composed of simpler components (such as 32-bit registers). This is partly for maintainability, but mostly because Parsim is more efficient when simulating fewer components.

3.2.3 Simulation Structure

In our simulators, the processor continuously issues memory read and write requests, pausing occasionally to send a message from the memory hierarchy to the network. In addition, a message will periodically arrive at the node from the network, at which point the network interface interrupts the processor to notify it of the message. In the cache–network simulator, the processor then reads the data from the network interface's message queues and writes it to memory, while in the primary memory–network simulator, the processor sets up registers in the direct memory access (DMA) controller, which then copies the message directly to primary memory. The average incoming and outgoing message rates (quantity of data per unit time) are equal and specifiable.

Figure 3.3 shows the path that incoming messages take on the cache–network simulator. First, messages enter the network interface from the interconnection network ($\boxed{1}$). The processor then reads the data from the network interface into registers[3] using ordinary memory read instructions ($\boxed{2}$). The CPU writes the data into memory (via a write buffer not pictured in Figure 3.3) in preparation for the user's code to access it ($\boxed{3}$). And, finally, the user's code reads the data into registers as it would any other data ($\boxed{4}$).

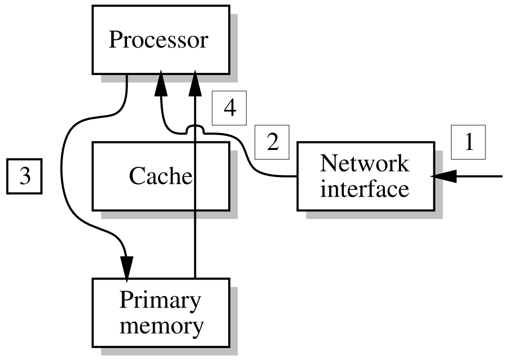


Figure 3.3: Cache–network incoming message path

Sending a message from CPU registers to the network (Figure 3.4) is the reverse process. The user's code writes the message into a buffer ($\boxed{1}$). (The gray lines in the figure represent the case where some or all of the data is displaced from the cache into memory.) Then, the CPU reads the message from the buffer ($\boxed{2}$) and sends the data to the network interface ($\boxed{3}$), which forwards it into the network ($\boxed{4}$).

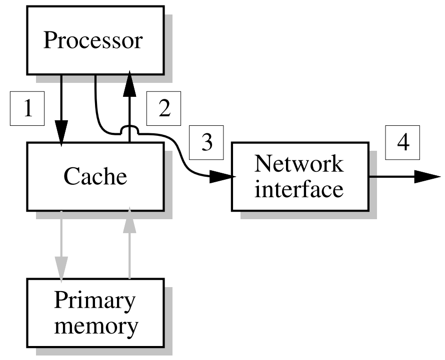


Figure 3.4: Cache–network outgoing message path

The primary memory–network simulator takes a different approach to moving data to and from the network. Message reception is shown in Figure 3.5. After a message enters the network interface ($\boxed{1}$), it is written to primary memory ($\boxed{2}$). When the user's program reads the received data into registers, it will miss in the cache and have to load the data from primary memory ($\boxed{3}$ & $\boxed{4}$).

As on the cache–network simulator, sending a message on the memory–network system (Figure 3.6) is the reverse of receiving one. The message must first be written to primary memory ($\boxed{1}$). From there, the message is sent to the network interface ($\boxed{2}$) and the network ($\boxed{3}$).

[3] But not into the cache; the addresses mapped to the network interface are never cached

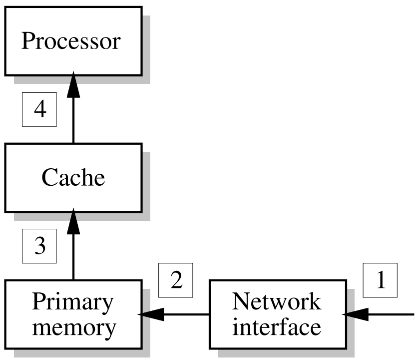


Figure 3.5: Memory–network incoming message path

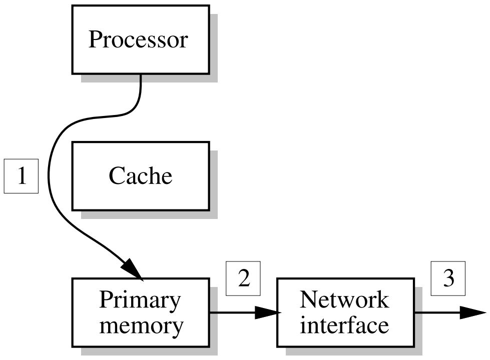


Figure 3.6: Memory–network outgoing message path

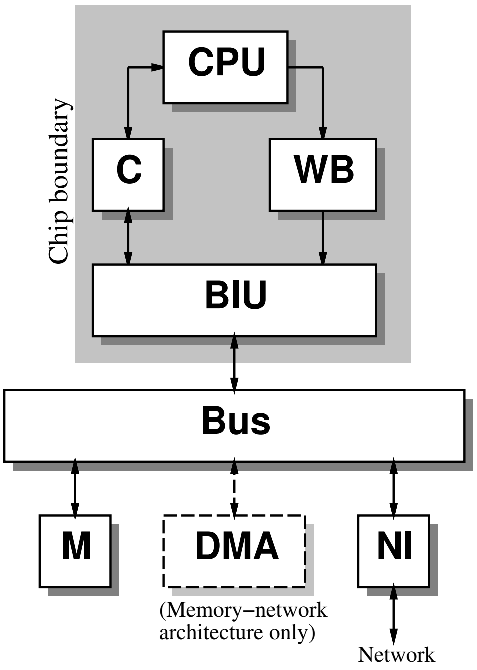
The logical hardware structure of all three versions of the simulator is shown in Figure 3.7. The CPU, cache, write buffer, and bus are based specifically on the DECchip 21064-AA implementation [14] of the Alpha architecture [13]. In addition to the modules shown in Figure 3.7, we simulate a *network fringe* (Figure 3.8). The network fringe embodies the actions of the remainder of the multicomputer (everything external to the node). Specifically, it sends messages to the network interface at a specified constant rate and accepts any messages sent by the node.

The simulators accurately model component and system functionality. But because all simulation-based research must make tradeoffs between accuracy and control and between accuracy and real-time performance, we made the following simplifications:

- Non-memory instructions execute in zero time.
- There is no operating system or run-time system.
- The surrounding parallel machine is modelled as a network fringe, which accepts and occasionally injects messages.

Those simplifications are reasonable because:

- Non-memory instructions have little effect in a study concerned exclusively with memory hierarchy performance.



CPU = Central Processing Unit
C = Cache
WB = Write Buffer
BIU = Bus Interface Unit
Bus = Bus
M = Memory
DMA = Direct Memory Access controller
NI = Network Interface

Figure 3.7: Logical structure of a simulated multicomputer node

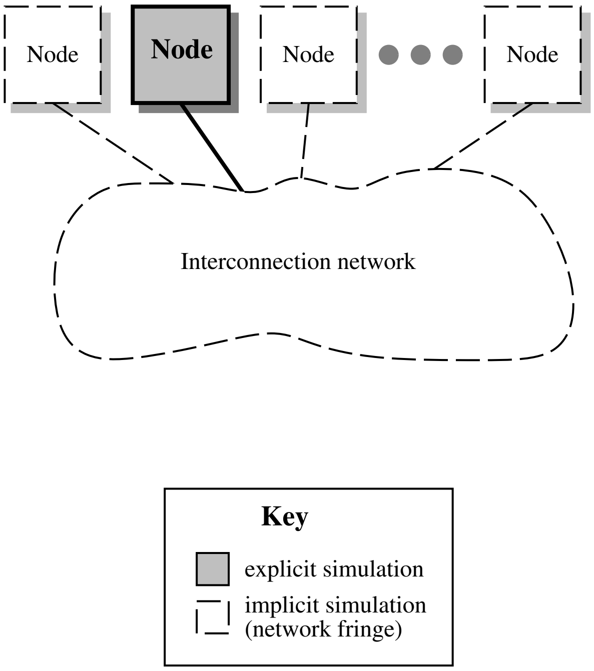


Figure 3.8: Network fringe

- Omitting operating system effects removes nondeterminism from cache performance and makes results consistent across runs.
- Surrounding a single node with a network fringe focuses the study on the effect of message handling by providing the flexibility to fine-tune the message rate and message length.

3.2.4 Simulation Structure

The CPU module simulates a 64-bit CPU. The translation lookaside buffer (TLB) and instruction cache hit rate is 100%.[4] The CPU module repeatedly performs the following operations:

1. Dequeuing a memory reference from the trace file, propagating it through the memory hierarchy
2. Responding to interrupts (incoming messages) by moving a message from the network to the memory hierarchy
3. Periodically sending a message from the memory hierarchy to the network

Note that only memory instructions are simulated; other instructions are ignored. Furthermore, the memory locations selected for operations 2 and 3 are uniformly distributed.

The cache module represents an on-chip physically-addressed word-addressable direct-mapped data cache modelled after the 21064-AA's (Figure 3.9). The cache uses a read-allocate, write-no-allocate scheme for misses. Memory locations greater than a specified address are uncacheable and therefore are guaranteed to miss in the cache. The cache module performs the operations listed in Table 3.1.

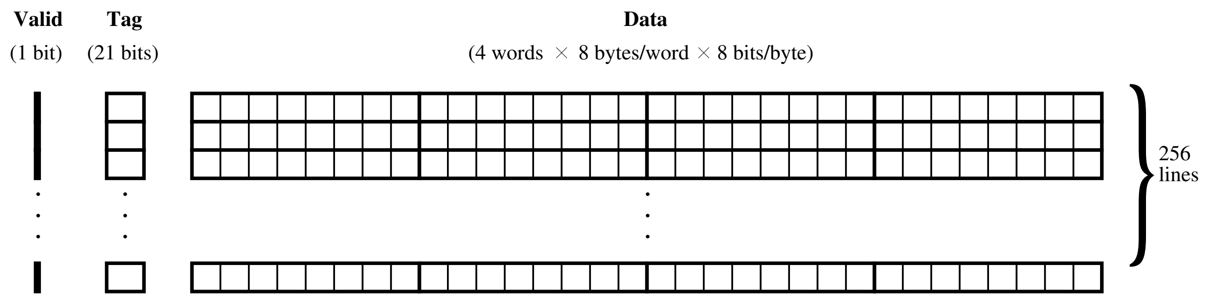


Figure 3.9: Layout of the 21064-AA's data cache

The write buffer is flushed on a read miss. This is necessary to ensure that the value read is up to date. While it would be possible for an implementation to permit the cache to read data directly from the write buffer, for simplicity, the 21064-AA does not.

The bit widths of the tag, index, and offset fields vary with cache size. Larger cache sizes increase the number of index bits and decrease the number of tag bits (*offset* is fixed at 2 bits), but maintain the equations:

$$tag + index + offset = 31$$

[4] Like many current CPUs, our CPU module uses separate instruction and data caches.

	Hit	Miss
Read	Send a data word from the cache to the CPU.	Fetch a cache line from memory.
Write	Modify a data word in the cache.	Do nothing.

Table 3.1: Cache module operations

and

$$2^{index} \text{ lines} \times \frac{2^{offset} \text{ words}}{1 \text{ line}} \times \frac{64 \text{ bits}}{1 \text{ word}} \times \frac{1 \text{ byte}}{8 \text{ bits}} \times \frac{1 \text{ MB}}{1024 \text{ bytes}} = \begin{array}{l}\text{cache}\\ \text{size (KB)}\end{array}$$

Like the cache module, the write buffer module is modelled after the 21064-AA's version (Figure 3.10). The write buffer module is implemented as a four-line-deep queue of cache lines. A mask bit corresponds to each word in the write buffer and specifies whether the word is valid (modified) or invalid (unmodified). Multiple writes to a location are merged in the write buffer. (See [14] for further details.)

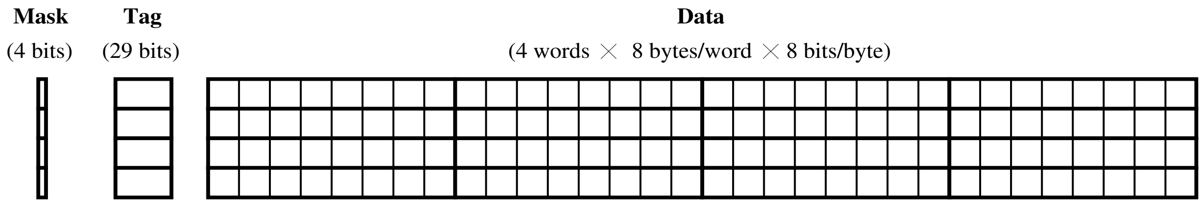


Figure 3.10: Layout of the 21064-AA's write buffer

The bus module models a non-pipelined bus with 128 data and 32 address lines, similar to the 21064-AA's. The bus interface module arbitrates fairly between the cache and the write buffer for control of the bus. It passes requests onto the bus, and—for a read request—returns the data read to the cache. Note that because the bus is 128 data bits wide but a cache/write buffer line is 256 bits wide (4 words × 64 bits/word), it takes two bus cycles to transmit a line.

The memory and memory controller modules sink write requests and return arbitrary data for read requests.

The network interface transmits messages between the bus and the router (really the network fringe). It accepts commands from the CPU by providing memory-mapped message queues and registers. When sending messages to the router on the cache–network simulator's network interface module, the first word written to the network interface's registers must contain the destination node, and the second word must contain the number of data words to follow. When receiving messages from the router, the network interface interrupts the CPU to alert it of message arrival. It then reads data from the router until its message queue is full, at which point it tells the router to stop sending. The CPU reads data out of the message queue. The network interface module in the primary memory–network simulator contains a DMA controller. It accepts commands from the CPU to send data from a given memory location to the router and to copy data from the message queue to memory.

The network fringe module sinks messages sent to it and periodically sends messages to the network interface on up to four physical channels. It models applications' communication patterns using a Poisson distribution and maintains a specified average message rate. Pseudocode for all the modules described in this section is given in Appendix A.

3.3 Trace Data

No single application is representative of the widely-varying memory access patterns different applications use. Therefore, we use memory traces from several different applications. We selected application traces based on the following criteria:

- scientific nature
- programming language
- locality
- size

The applications are taken from well-known benchmark suites (SPEC [15, 16] and SPLASH [31]). They are written in different languages (C and Fortran). They exhibit a range of localities. And they each make enough memory references to provide meaningful data, but not so many as to take an inordinate amount of time to simulate. We use the same trace files for both the uniprocessor and multicomputer simulators—to do otherwise would confuse effects caused by a node architecture and artifacts of an application's implementation. Trace data was generated using QPT [2, 26].

Information about the applications is given in Table 3.2. The column labelled "Memory references" represents the number of `LOAD`s and `STORE`s issued dynamically. "Working set size" was calculated by:

1. Running the memory traces through dineroIII[5] using various-sized fully-associative caches with LRU replacement
2. Plotting the miss rate against the cache size
3. Taking the knee in each curve as the corresponding application's working set size.

Descriptions of the traced applications [9, 15, 31] follow:

`barnes` — `barnes` uses Barnes-Hut's hierarchical N-body algorithm to model point-masses exerting gravitational forces on each other in three dimensions. The algorithm specifies that clusters of distant point-masses are treated as a single point-mass, which significantly reduces the amount of necessary computation. We specified a single-processor run and used the SPLASH distribution's input parameters (most notably, 128 point-masses simulated for 11 time steps).

[5] Available from `ftp://ftp.cs.wisc.edu/markhill/Misc/dineroIII.3.4.tar`

Application	Benchmark suite	Language	Memory references	Working set size
`barnes` ("Barnes-Hut")	SPLASH	C	128,299,383	32KB
`doduc`	CFP92	Fortran	113,957,528	64KB
`ear`	CFP92	C	256,352,000	8KB
`fpppp`	CFP92	Fortran	294,101,082	16KB
`mdljsp2`	CFP92	Fortran	64,567,619	128KB
`pthor`	SPLASH	C	58,133,474	64KB
`xlisp` ("Li")	CINT92	C	63,666,825	64KB

Table 3.2: Application traces

`doduc` — `doduc` uses Monte Carlo techniques to simulate the time evolution of a thermohydraulical modelization of a nuclear reactor's component. It is intended to be typical of ECAD and high-energy physics applications. We used the "small" input set.

`ear` — `ear` models the propagation of sound in the human cochlea. It inputs a Macintosh MacRecorder sound file sampled at 22 kilohertz (kHz) and outputs a picture called a *cochleagram*, which shows the firing rate of the inner hair cells in the cochlea in response to the sound in the sound file. We used the "short" input set.

`fpppp` — The two electron integral derivative is a style of computation used by the GaussianXX series of quantum chemistry programs. `fpppp` solves the two electron integral derivative for some input number of atoms. We traced a 5-atom run.

`mdljsp2` — Molecular dynamics programs are used to test theories of interatomic potential, and `mdljsp2` specifically evaluates the validity of the Lennard-Jones potential. It simulates a number of atoms (500) interacting in an idealized Lennard-Jones potential and uses statistical mechanics to calculate position, velocity, energy, and pressure to track each atom individually. The results of simulation could then be compared to experimental results to establish the accuracy of the potential. Like `fpppp`, `mdljsp2` is intended to be representative of quantum chemistry applications. We used the "short" input set.

`pthor` — Given the description of a digital logic circuit and its input, `pthor` simulates the operation of that circuit using a variant of the Chandy-Misra distributed-time algorithm [5]. As an event-driven simulator, `pthor` has much in common with Parsim (Section 3.2.2); both simulate arbitrarily complex functional blocks interconnected by wires. But while Parsim's clock is global, `pthor`'s is distributed (an aggressive optimization). That is, the various functional blocks do not generally observe the same simulated time. We simulated the `risc` circuit from the SPLASH distribution.

`xlisp` — `xlisp` is a small LISP interpreter. When used in SPEC, the interpreter's input is an implementation of the N-queens problem (Find all arrangements of N queens on an $N \times N$ chessboard such that no queen can take any other queen) based on that in [36, pp. 183, 367–368]. We specified seven queens.

3.4 Simulation Parameters

Variable	Description	Values	
Node architecture	Manner in which message traffic is handled	"cache" "memory" "uniprocessor"	
Message rate	Number of words sent and received per processor clock	0.0010 0.0020 0.0100 $\boxed{0.0200}$ 0.0400 0.0600 0.0800 0.1000	
Cache size	Size of cache in bytes	8KB $\boxed{16\text{KB}}$ 32KB 64KB	
Long message percentage	Percentage of messages (in number and volume) considered "long" ("long"≡128 words, "short"≡4 words)	#	Vol.
		0%	0%
		1%	24%
		2%	40%
		3%	50%
		4%	57%
		10%	78%
		20%	89%
		$\boxed{25\%}$	$\boxed{91\%}$
		50%	97%
		75%	99%
		100%	100%

Table 3.3: Simulation parameters; defaults are boxed

Table 3.3 lists the four simulation parameters we varied in each experiment. Entries marked as "default" were used except where otherwise noted. The following sections detail the simulation parameters and the range of values.

3.4.1 Node Architecture

We ran all our experiments on a cache–network, primary memory–network, and uniprocessor simulator, as described previously in this chapter.

3.4.2 Message Rate

In order to examine the relationship between message traffic and the cache miss rate, we varied the rate at which message data are sent between the processor and network. In this thesis, "message rate" refers to the average number of incoming+outgoing words per processor clock. Messages are introduced with a Poisson distribution. That is, the time intervals between messages are uniformly distributed while maintaining a given message rate.

Data derived from [11] (Appendix B) show that for the machines and applications examined in that paper, the message rate varies from 0.00004 to 0.00674 64-bit words per clock, with a mean of 0.00114. However, those numbers include the time spent executing non-memory instructions, while our simulations execute exclusively memory instructions. In other words, the numbers calculated from [11] are unrealistically low for a memory-only study. Hence, we adjusted our range of message rates upward. The message rates used emphasize "interesting" regions of the data—where the curves are not flat—and were selected based on preliminary results.

Preliminary results also show that there is a maximum message rate the two memory-network interfaces are capable of sustaining. At message rates upwards of approximately 0.1000 words/clock, the simulated cache–network and—to a lesser extent—memory–network systems spend so much time handling messages, they can make no other progress. This provides a practical lower bound on current node architectures communication granularity. Applications that attempt to transmit more frequently than 1 word for every 10 clocks spent in the memory hierarchy (very roughly, 1 word per 26 CPU instructions[6]) are likely to spend an inordinate amount of time waiting on message-handling routines and the cache refills they cause.

3.4.3 Cache Size

We experimented with a variety of cache sizes in order to forecast performance for future microprocessors. We simulated an 8KB cache because the 21064-AA—on which most of the components in our simulated microprocessor are based—uses an 8KB cache. But because modern microprocessors have ever larger primary data caches—16 KB has already become common—we also simulated 16KB, 32KB, and 64KB caches. We did not simulate a second-level cache.

3.4.4 Long Message Percentage

We experimented with both long and short messages to determine the effect of message length on the cache miss rate. Because they are allocated in contiguous memory locations, long messages give near-perfect spatial locality. In contrast, sequences of short messages generally do not. Because of its spatial locality, a single long message should displace fewer cache lines than an equivalent volume of short messages, and is therefore expected to be less harmful to cache performance.

[6] Based on data in [14, 19] and assuming a split 32 KB cache and an average execution time of one cycle per instruction (CPI) for non-memory instructions

To quantify the impact of short and long messages, we simulated different mixes of two message lengths (short and long). As with message rate, the specific long message percentages we chose emphasize "interesting" regions of data and were selected based on preliminary results.

3.5 Dependent Variables

Cache misses is the dependent variable used in all of the graphs in Chapter 4. Most of those graphs normalize the number of cache misses to that observed on the uniprocessor simulation. Where appropriate, absolute misses are also used. Normalized numbers represent how many times slower an application can be expected to run in the presence of message traffic.

For the purposes of this study, cache misses is a superior metric both to execution time and cache miss rate. Execution time is technology-dependent, tying findings to specific choices of component speeds. Cache miss rate, while commonly used in cache studies, is inappropriate for this study because the number of memory references is variable; message handling introduces additional memory references. Cache misses, unlike miss rate, does not depend on the total number of memory references and therefore provides better intuition about the data.

In Chapter 4, cache read misses that access primary memory are considered "misses," and all other cacheable memory operations are considered "hits." Most notably, cache write misses are considered hits because the write buffer and write no-allocate cache hide memory latency, thereby making write miss response time approximately equal to that of a write hit. Uncacheable operations (reads and writes to memory-mapped I/O locations) are deemed I/O and are therefore considered neither hits not misses. Note that references made to primary memory as a result of network traffic (viz. storing incoming messages to memory) are considered regular memory operations.

3.6 Component Speeds

For our experiments, we assigned the following delays—which we believe are reasonable for a 200MHz DEC Alpha-based system (5 ns cycle time)—to the components described in Section 3.2:

cache access	1 CPU cycle
write buffer access	1 CPU cycle
bus cycle	2 CPU cycles
memory access	10 CPU cycles
network cycle	$\frac{1}{2}$ CPU cycle
router in→out latency ...	2 CPU cycles

All other components have a latency of 1 cycle on their interfacing component's clock. Note that write buffer accesses are performed in parallel with cache accesses.

Chapter 4

Simulation Results and Analysis

4.1 Base Misses

We measure the impact of message traffic on the cache by comparing multicomputer and uniprocessor cache misses. Figure 4.1 plots the uniprocessor misses for each application. The graphs in Sections 4.2–4.4 are normalized to the misses shown in Figure 4.1 and therefore represent the increase in the number of multicomputer cache misses compared to the corresponding uniprocessor simulation. Note that uniprocessor simulations will always observe a lesser number of misses than multicomputer simulations because they perform an application's memory references uninterrupted by miss-causing message traffic.

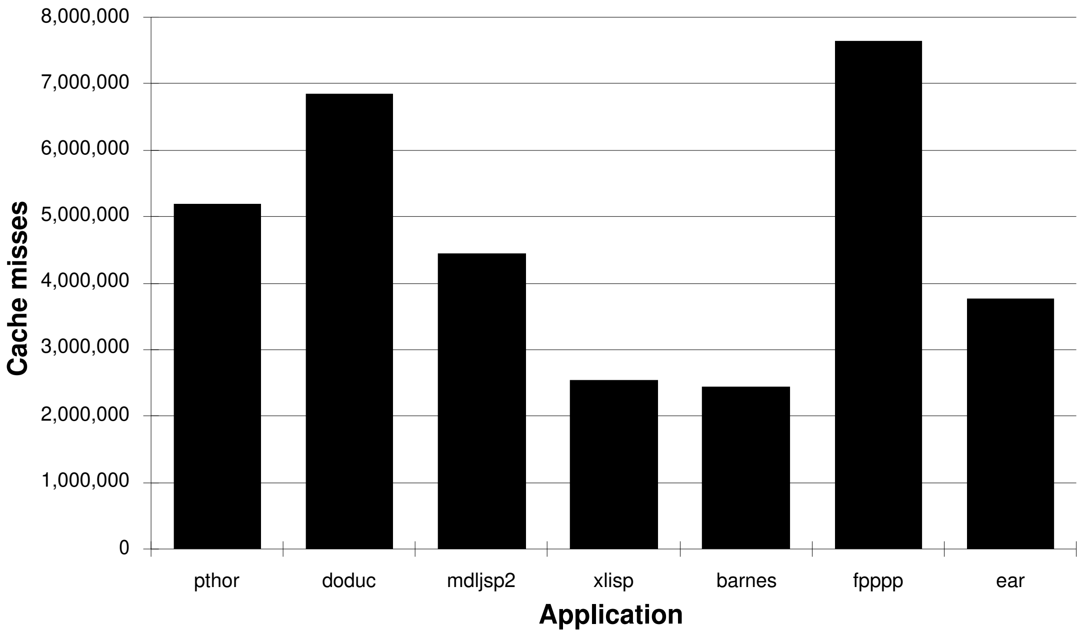


Figure 4.1: Uniprocessor cache misses

4.2 Misses versus Message Rate

The first experiment shows how sensitive the cache is to the volume of message traffic per unit time. The average message rate was varied from 0.0010–0.0800 words sent/received per processor clock. Figures 4.2–4.8 show the cache–network and memory–network systems' increase in misses for each application, normalized to the number of misses observed on the uniprocessor system. Figure 4.9 repeats that data on a single three-dimensional graph. Note that one of the abscissas is on a logarithmic scale.

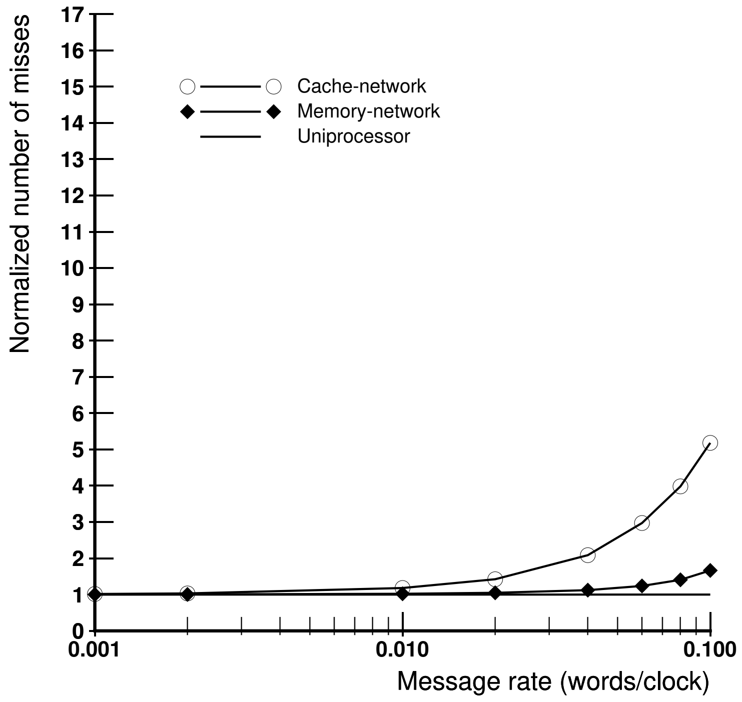


Figure 4.2: Normalized number of misses vs. message rate (`pthor`)

For all seven applications, the effects of an increased message rate are far more pronounced on the cache–network system than on the primary memory–network system. That result is in accordance with our intuition, as the cache–network system utilizes the cache for message handling while the primary memory–network system offloads message handling to the DMA controller. The difference between the two multicomputer systems is significant at high message rates, with the cache–network system observing many times more misses than the primary memory–network system.

Sensitivity to cache pollution is related to memory reference locality. Compare the height of each line in Figure 4.9 to the height of the corresponding bar in Figure 4.10, a plot of each application's uniprocessor cache miss rate. While the heights are not strictly inversely proportional, they are close enough to imply that locality is a factor in cache sensitivity to message traffic. That makes sense because cache lines displaced in a high-locality application are apt to be accessed again, thereby introducing more cache misses than would occur from an application with less locality.

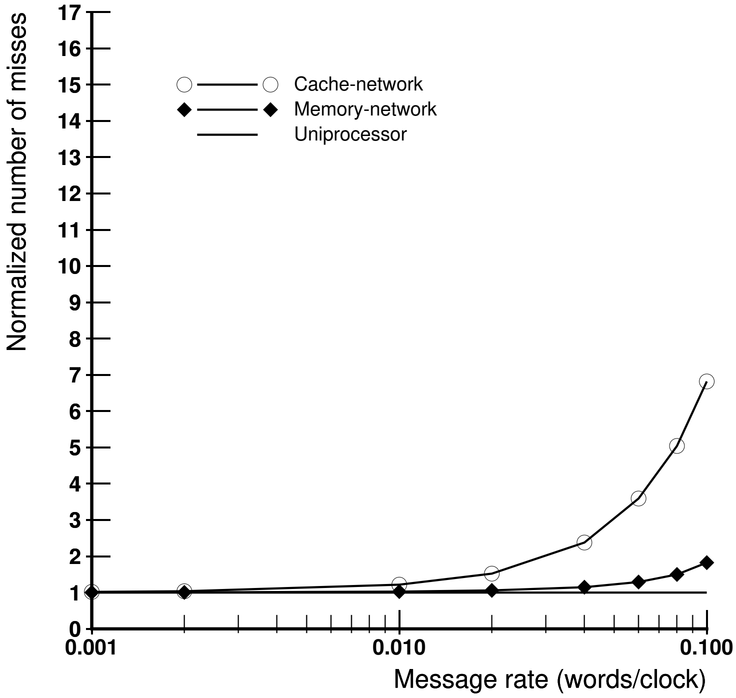


Figure 4.3: Normalized number of misses vs. message rate (`doduc`)

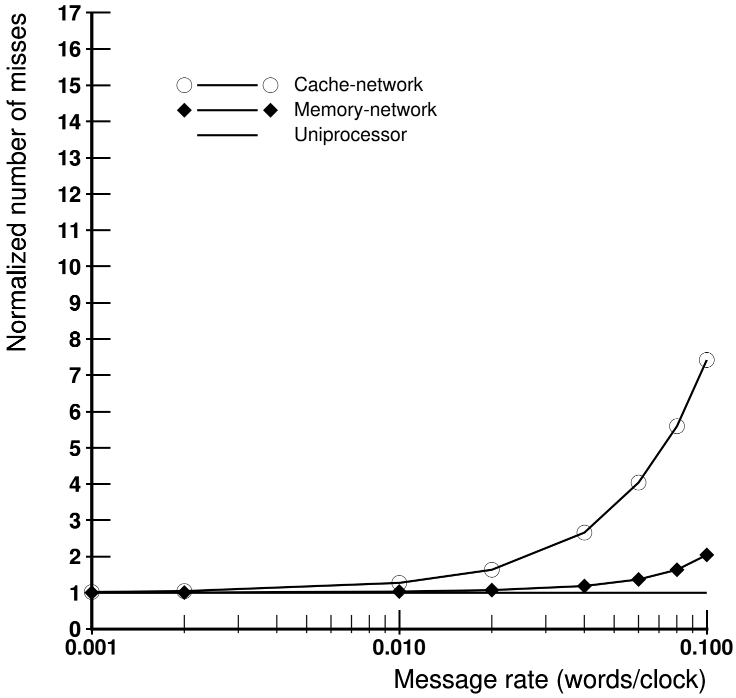


Figure 4.4: Normalized number of misses vs. message rate (`mdljsp2`)

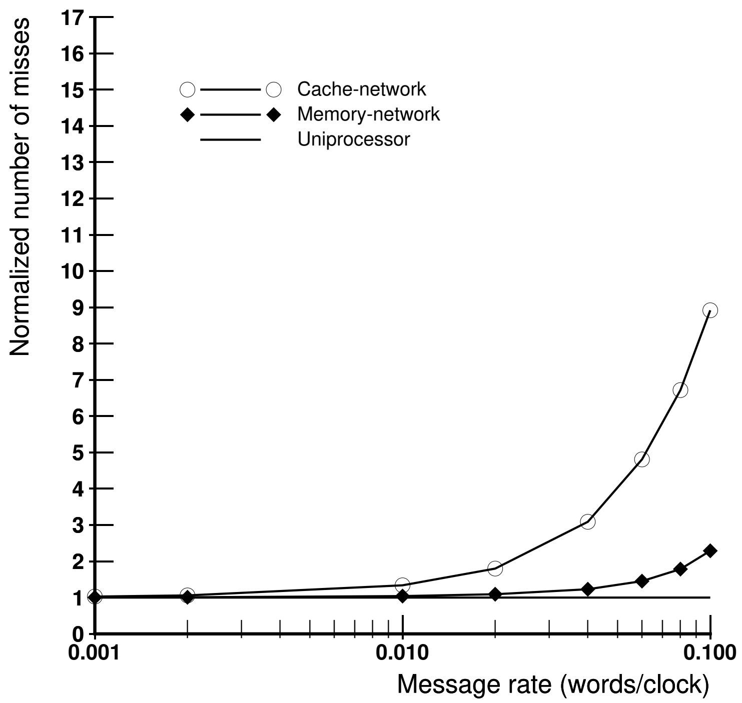


Figure 4.5: Normalized number of misses vs. message rate (`xlisp`)

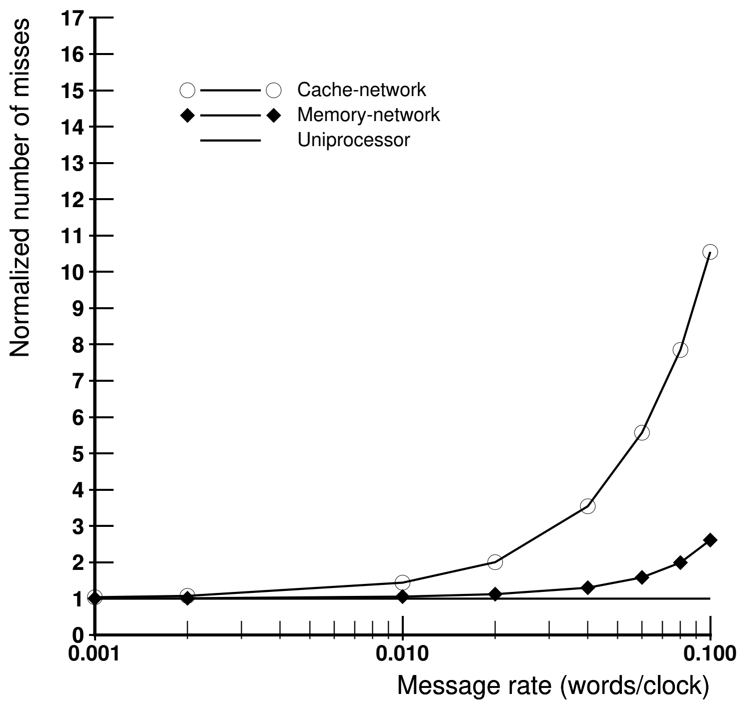


Figure 4.6: Normalized number of misses vs. message rate (`barnes`)

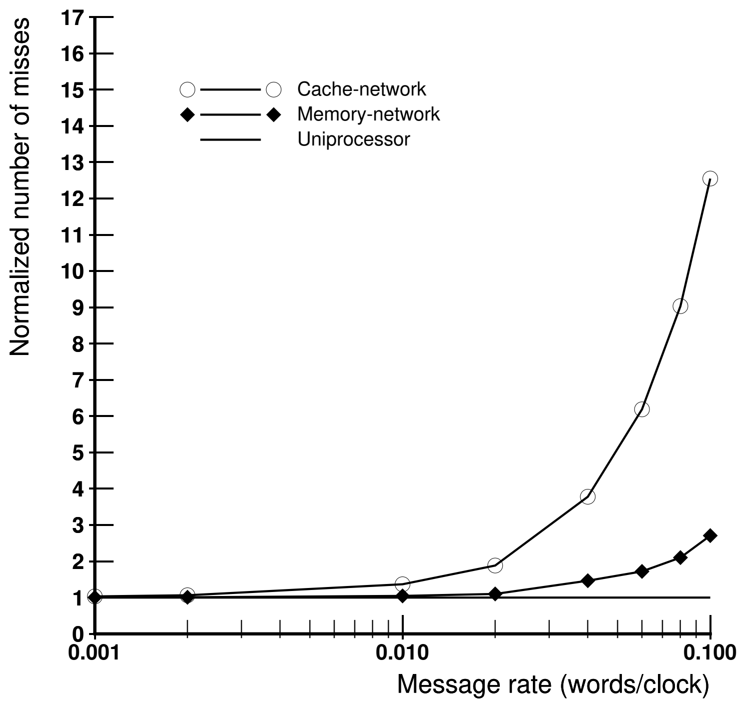


Figure 4.7: Normalized number of misses vs. message rate (`fpppp`)

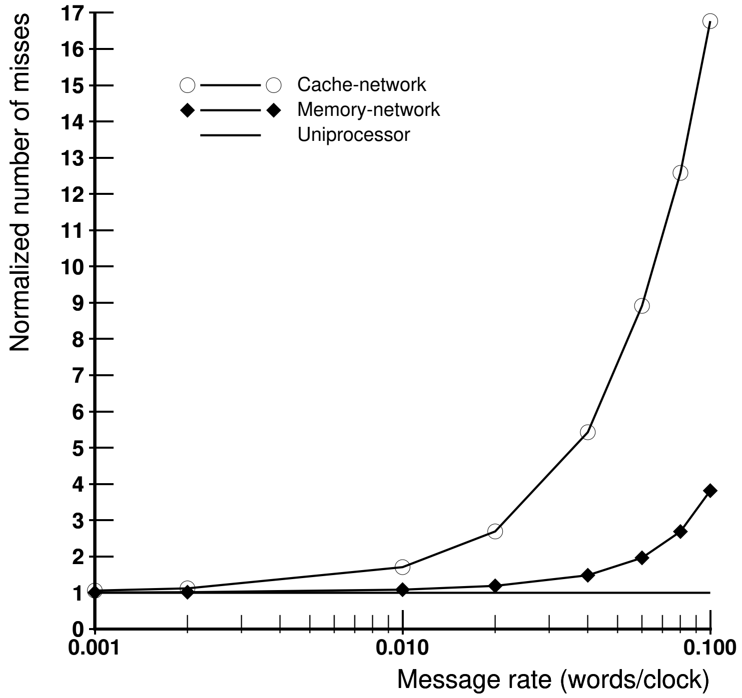


Figure 4.8: Normalized number of misses vs. message rate (`ear`)

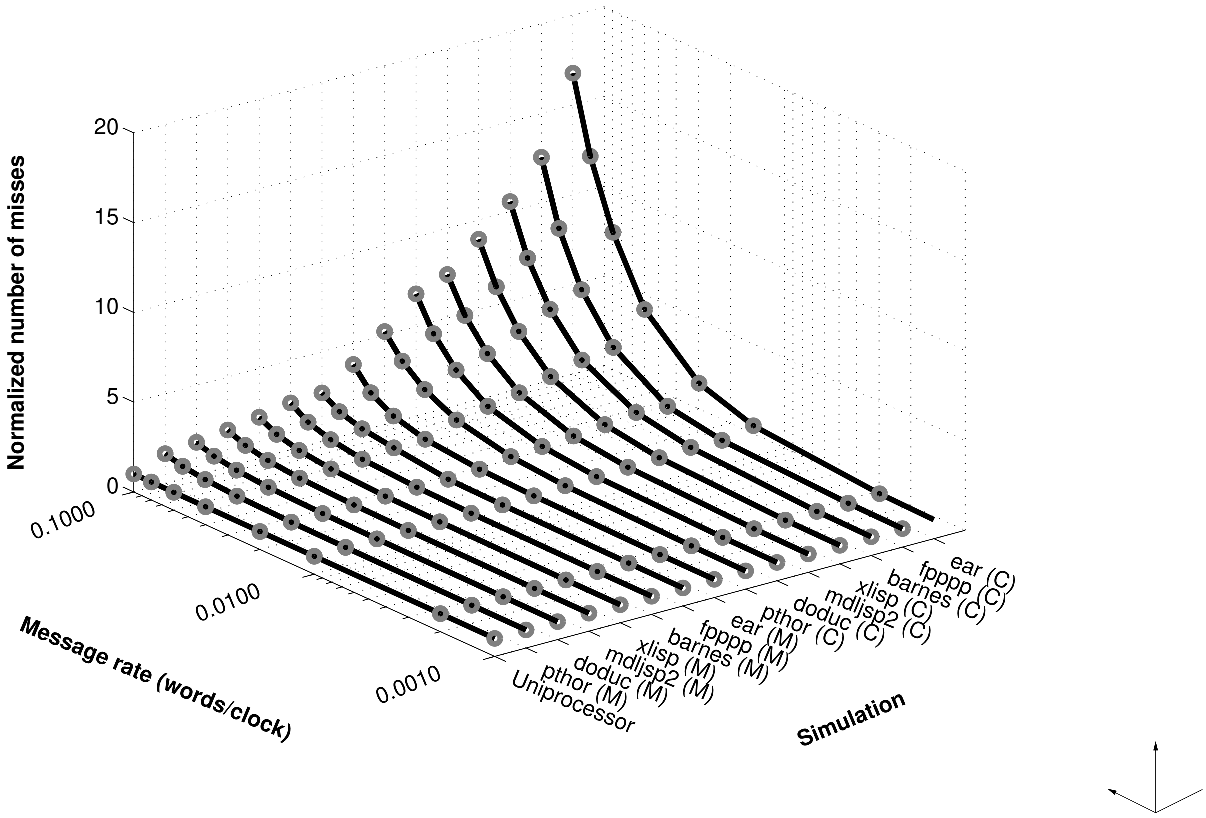


Figure 4.9: Normalized number of misses vs. message rate;
(C) indicates the cache–network system,
(M) the primary memory–network system

Data layout is another factor in cache sensitivity. The more an application's data is scattered throughout memory, the less likely it is that a single message will displace more than a few words used by the application.

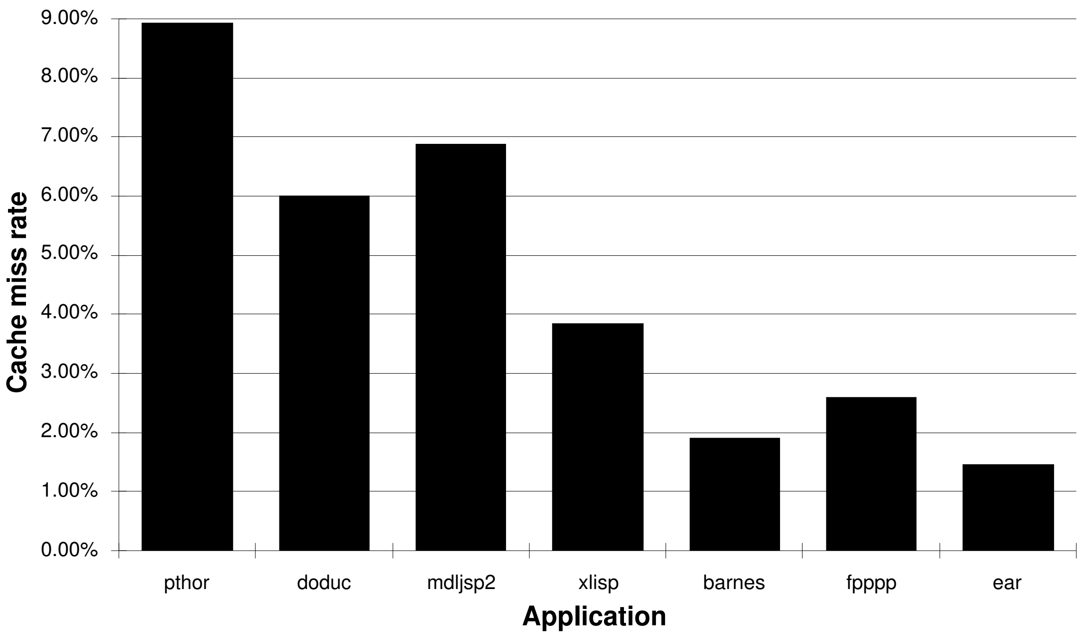


Figure 4.10: Uniprocessor cache miss rate

While Figures 4.2–4.9 accurately represent the cache–network system's worst-case performance[1], they do not differentiate between the two sources of cache misses: misses caused by accesses to displaced application data and misses caused by loading message data into the cache prior to transmission. Because data are transmitted from uniformly-distributed locations in memory, they are unlikely to be cache-resident and are therefore almost certain to miss in the cache. To differentiate misses caused by displaced application data from misses caused by the pre-transmission cacheing of message data, we present Figures 4.11–4.17, which portray the same data as Figures 4.2–4.8 except that the former subtract off misses caused by reading data into the cache in preparation for message sending (i.e. one miss for every cache line sent). This emphasizes exclusively misses caused by displaced application data. Figure 4.18 combines Figures 4.11–4.17 into a single three-dimensional graph. Note that one of the abscissas is on a logarithmic scale.

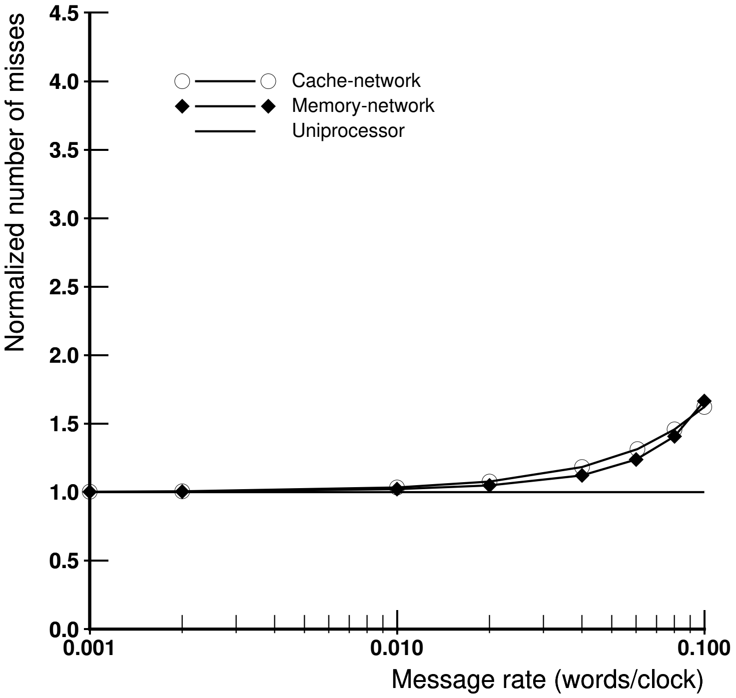


Figure 4.11: Normalized number of "non-sending" misses vs. message rate (`pthor`)

[1] Best case performance—where only cached data is transmitted—resembles uniprocessor performance and is therefore uninteresting from this study's perspective.

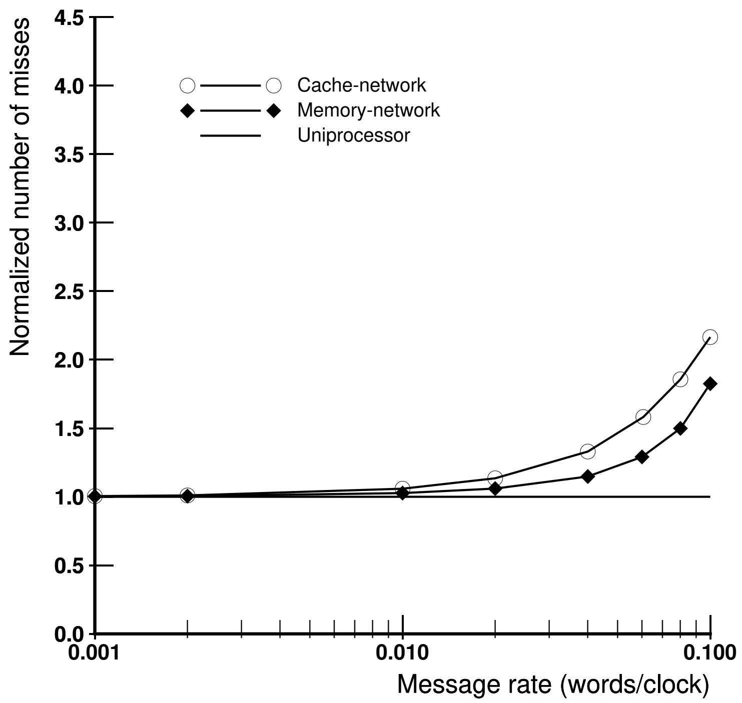


Figure 4.12: Normalized number of "non-sending" misses vs. message rate (`doduc`)

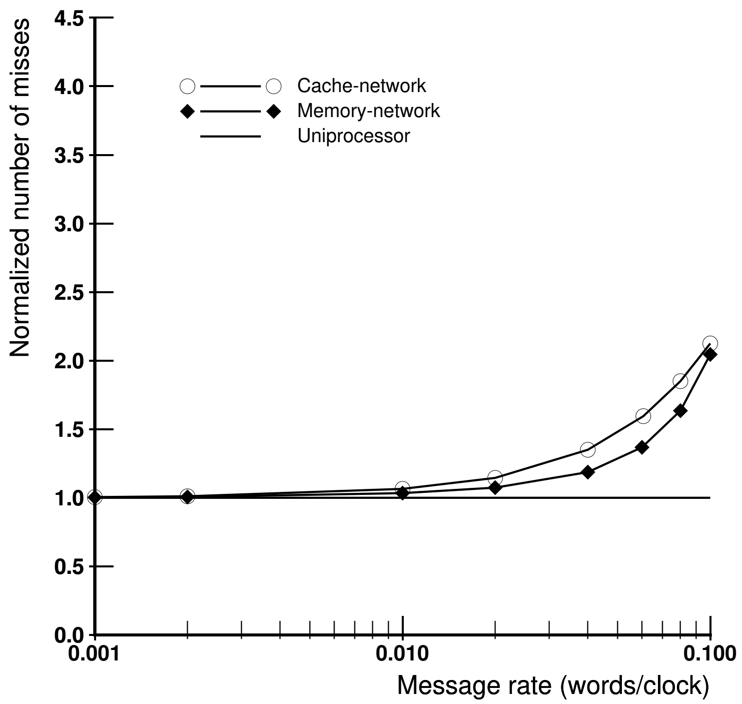


Figure 4.13: Normalized number of "non-sending" misses vs. message rate (`mdljsp2`)

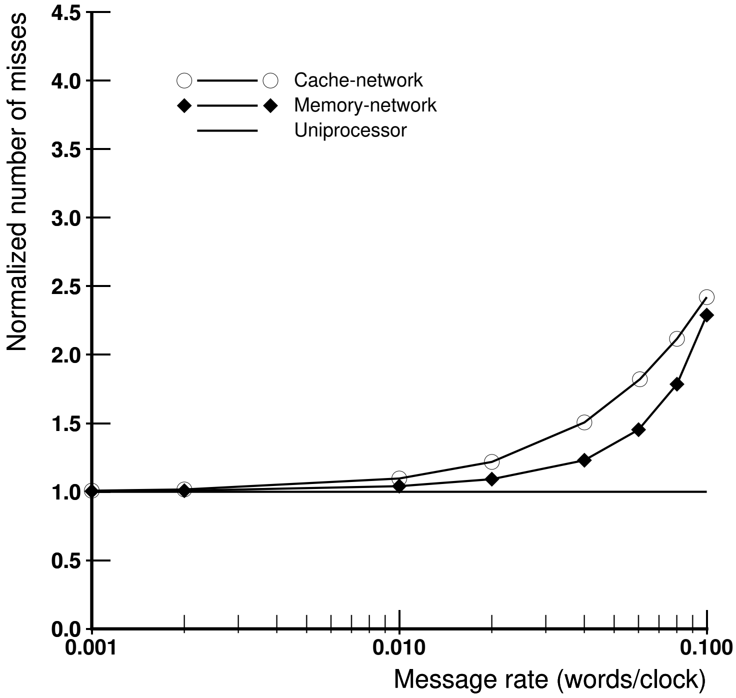


Figure 4.14: Normalized number of "non-sending" misses vs. message rate (`xlisp`)

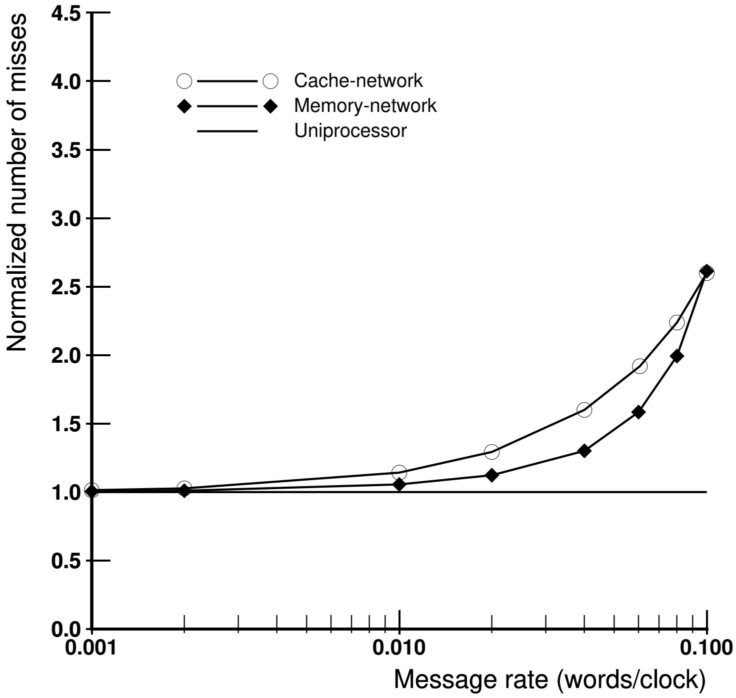


Figure 4.15: Normalized number of "non-sending" misses vs. message rate (`barnes`)

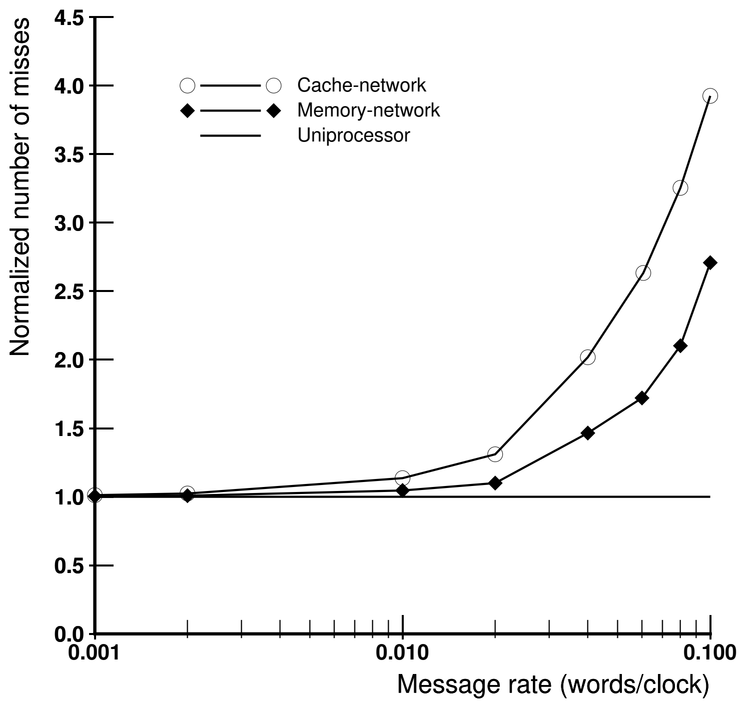


Figure 4.16: Normalized number of "non-sending" misses vs. message rate (`fpppp`)

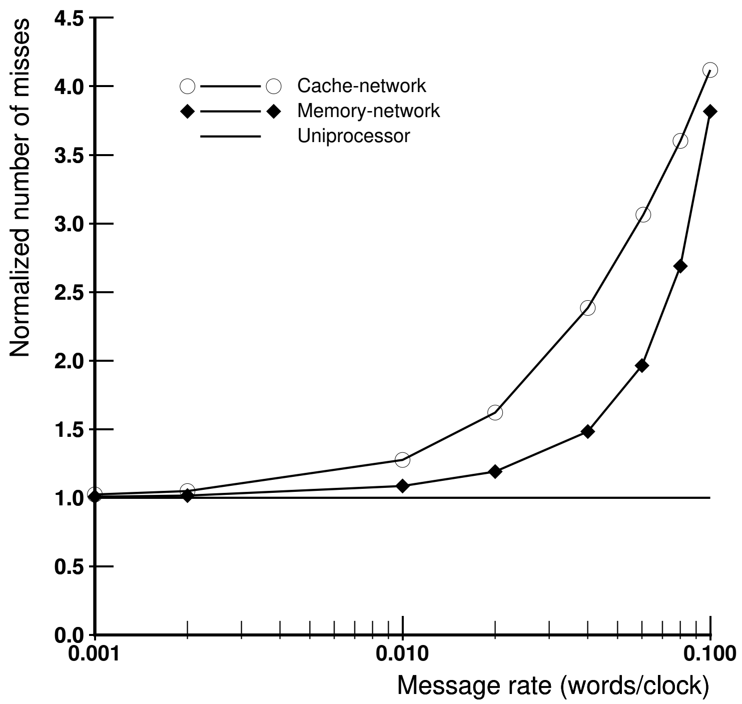


Figure 4.17: Normalized number of "non-sending" misses vs. message rate (`ear`)

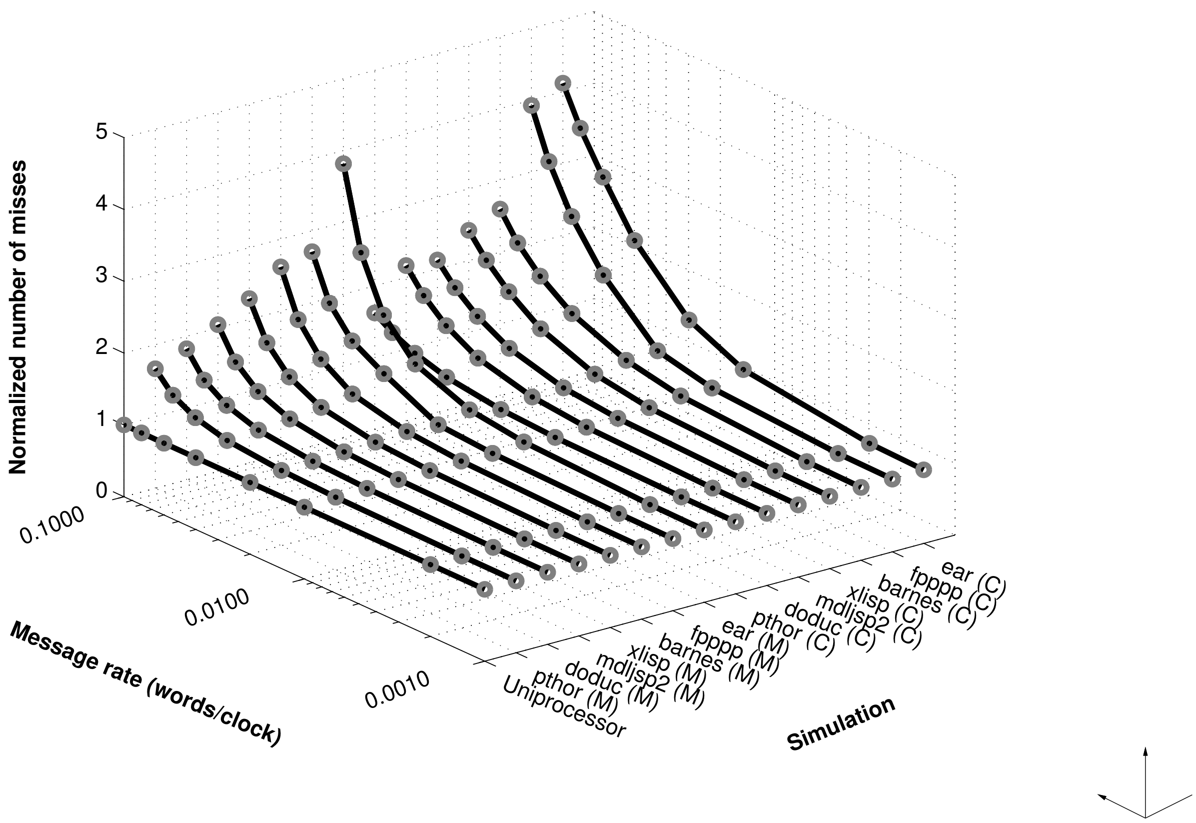


Figure 4.18: Normalized number of "non-sending" misses vs. message rate; (C) indicates the cache–network system, (M) the primary memory–network system

What is most striking about the data shown in Figures 4.11–4.18 is the proximity of the cache–network and primary memory–network curves. The implication is that the cache–network and primary memory–network systems perform displace approximately equal amounts of application data. The bulk of the cache–network system's cache misses from Figures 4.2–4.9 arise from loading message data into the cache—a necessary operation, but one that would be greatly reduced if only recently-used (implying cached) data were transmitted.

4.3 Misses versus Cache Size

Future microprocessors will almost certainly contain larger caches than current ones. Therefore, predicting the performance of forthcoming message-passing systems depends on the sensitivity of these changes to cache size. To determine that relation, we varied the cache size from 8KB to 64KB and held all other simulation parameters at the default values (see Table 3.3). Figures 4.19–4.25 plot the normalized number of cache misses observed by each application, and Figure 4.26 plots the normalized number of cache misses observed by *all* applications on a single, three-dimensional graph.

Figures 4.27–4.29 plot the *unnormalized* number of cache misses to put Figures 4.19– 4.26 in perspective. Note that for legibility, Figures 4.27–4.29 plot increasing cache sizes back-to-front, while Figure 4.26 plots them front-to back.

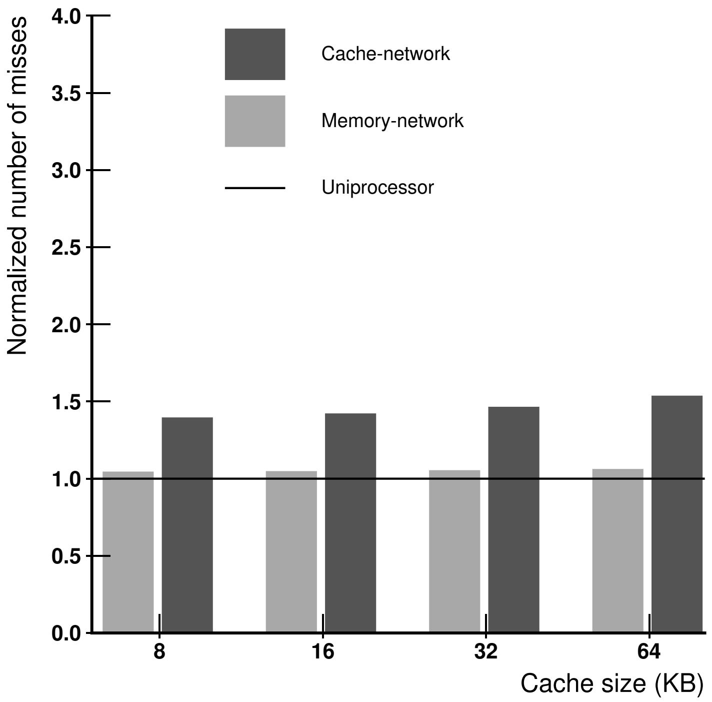


Figure 4.19: Normalized number of misses vs. cache size (`pthor`)

For all seven applications, each multicomputer's normalized number of misses *increases* with cache size (with the exception of `ear` from a 32KB–64KB cache—most likely a threshold effect).

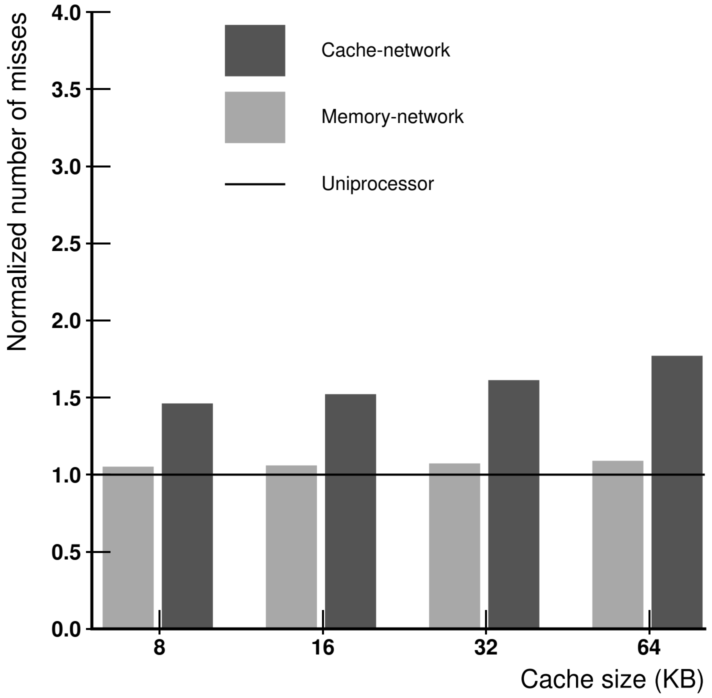


Figure 4.20: Normalized number of misses vs. cache size (`doduc`)

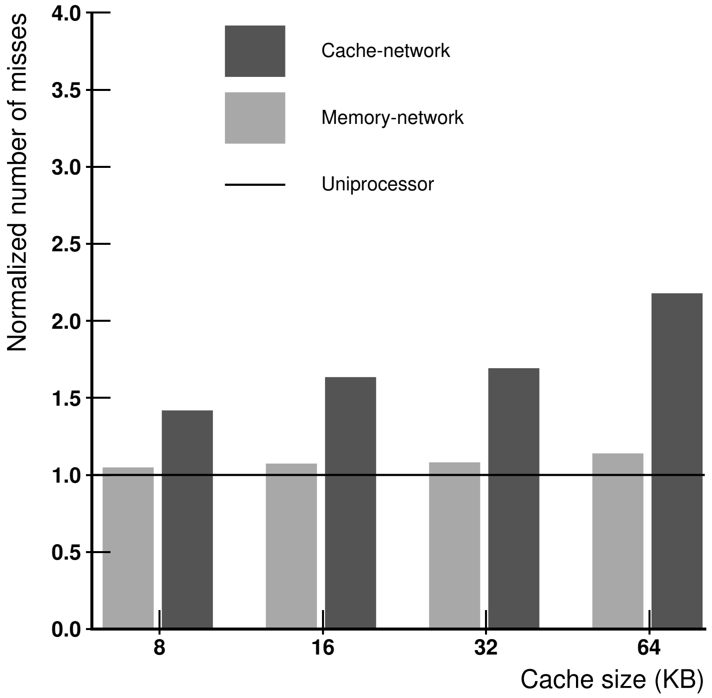


Figure 4.21: Normalized number of misses vs. cache size (`mdljsp2`)

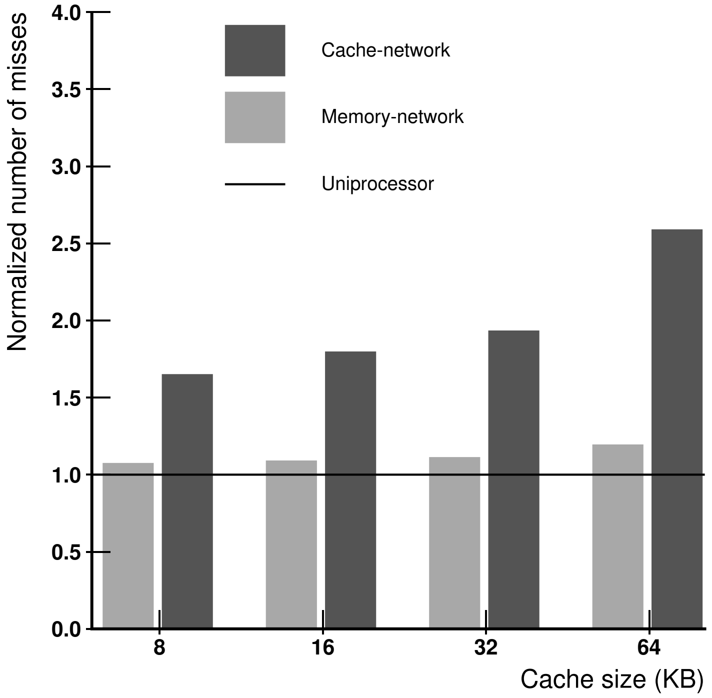


Figure 4.22: Normalized number of misses vs. cache size (`xlisp`)

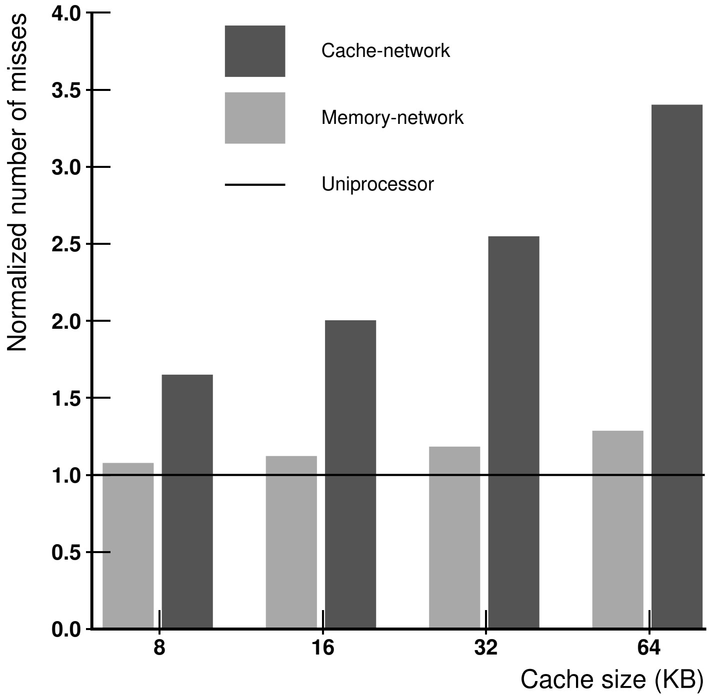


Figure 4.23: Normalized number of misses vs. cache size (`barnes`)

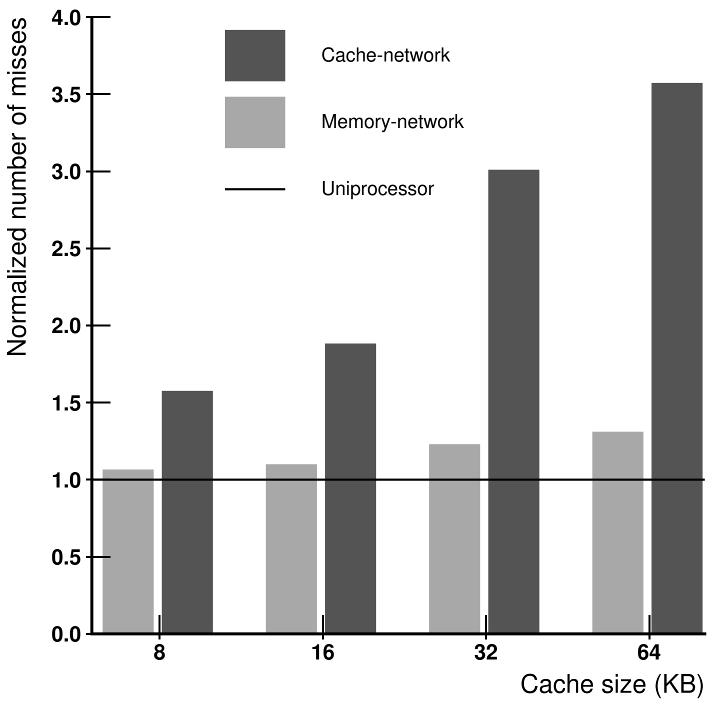


Figure 4.24: Normalized number of misses vs. cache size (`fpppp`)

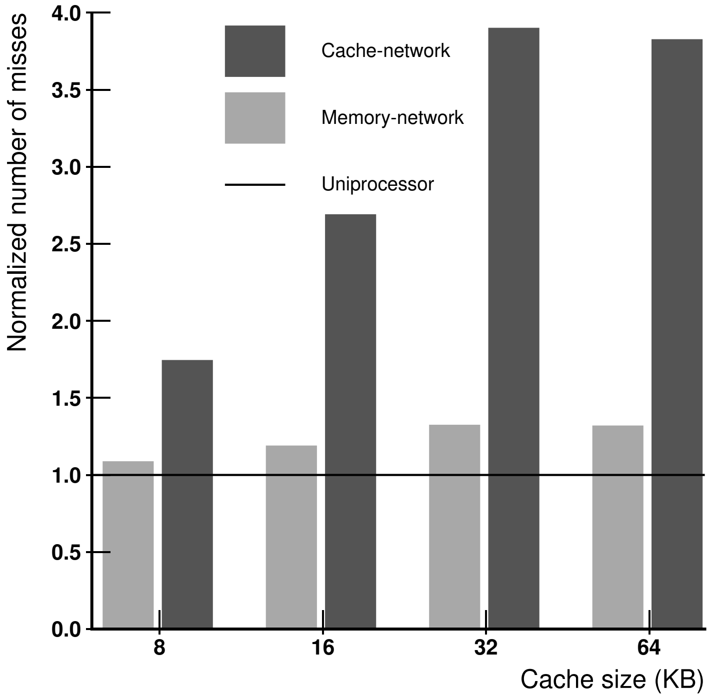


Figure 4.25: Normalized number of misses vs. cache size (`ear`)

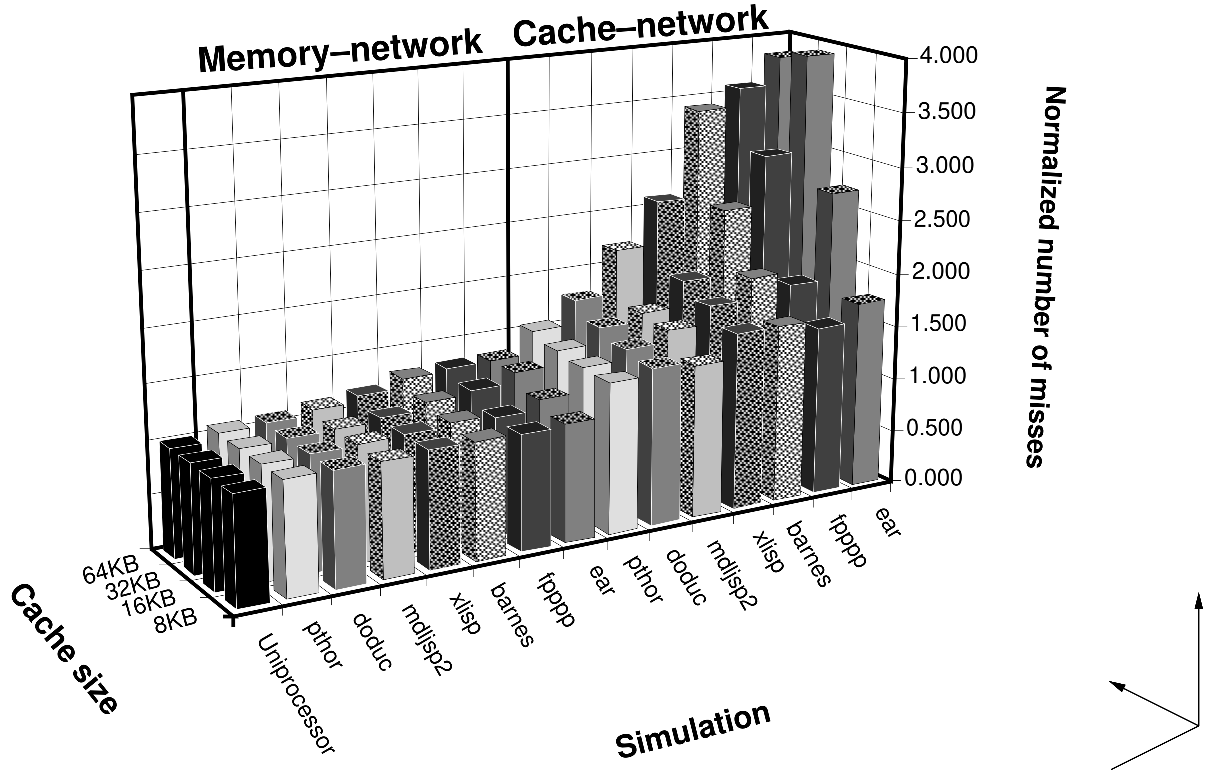


Figure 4.26: Normalized number of misses vs. cache size

Of course, increasing the cache size decreases the absolute number of misses for all applications and node architectures. The intuition, then, is that the increase in cache misses incurred by communication is magnified by the lower number of misses.

As Figure 4.26 shows, the primary memory–network system observes essentially the same number of misses as the uniprocessor system. In the worst case (i.e. the greatest increase in misses)—`ear` with a 32KB cache—36% more misses were observed on the primary memory–network system than on the uniprocessor. In contrast, the cache–network system is extremely sensitive to message traffic for large cache sizes. In that system's worst case—also `ear` with a 32KB cache—290% more misses are observed than on the uniprocessor! Even the cache–network system's *best* case (`pthor` with an 8KB cache) shows a 40% increase in number of misses—even worse than the primary memory–network system's *worst* case mentioned above.

The reason the cache–network system consistently exhibits a greater increase in misses than the primary memory–network system is that the processor reads into the cache all words to be transmitted. While the read itself might miss in the cache, it may also displace cache lines used by the application for an additional performance penalty. Because the primary memory–network system uses DMA to transport messages directly from memory to the network interface, no additional cache misses are generated for message sends.

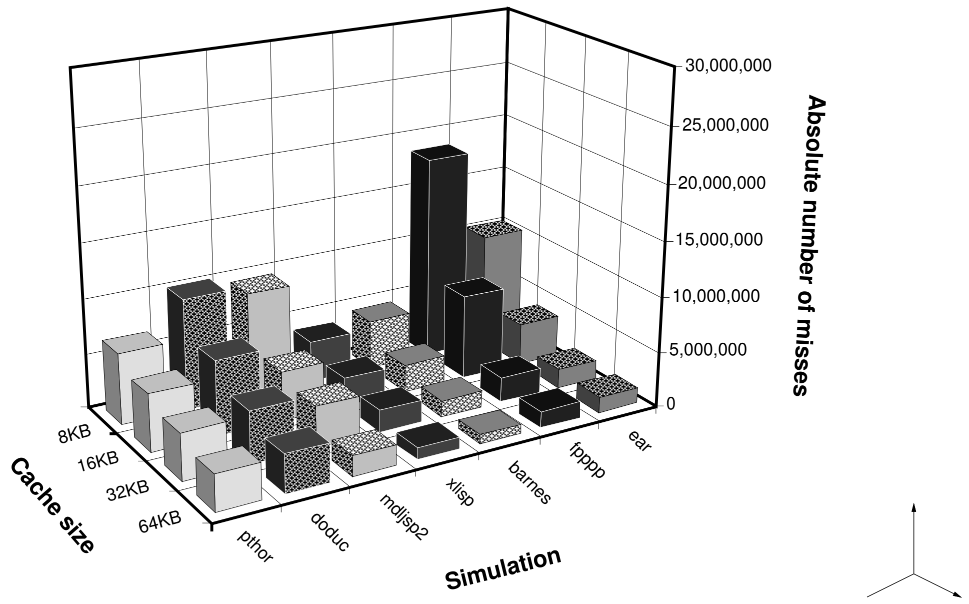


Figure 4.27: Absolute misses vs. cache size (uniprocessor)

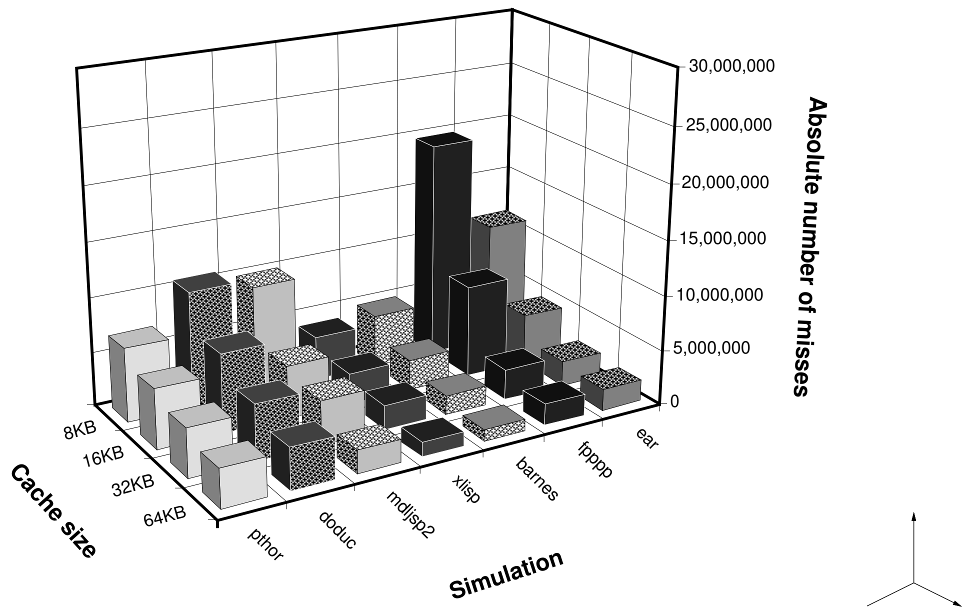


Figure 4.28: Absolute misses vs. cache size (memory–network)

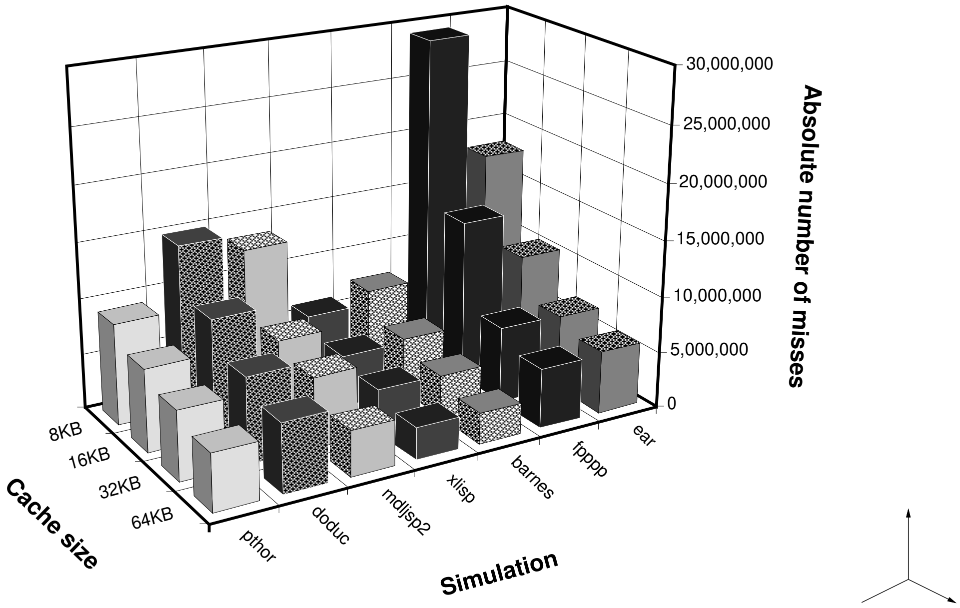


Figure 4.29: Absolute misses vs. cache size (cache–network)

4.4 Misses versus Message Size

Message-passing applications employ a large variety of message lengths [11]. Therefore, we investigated the effect of message size on cache performance. We expect message size to impact the cache for two reasons: First, a small number of long messages exhibits more spatial locality than a large number of short messages. Second, long and short messages tend to be used differently in message-passing applications: short messages for program control (e.g. for synchronization and broadcasting state information) and long messages for transporting program data structures. The difference in usage is that control message are generally read upon receipt, while data messages are read piecewise, with intermittent computation.

Our simulators represent that difference in usage—both in this section and all the others—by immediately processing control messages. To that effect, the primary memory–network system's CPU reads control messages upon arrival, but leaves data messages in memory. That is, it ignores steps 3 and 4 in Figure 3.5. On the cache–network system, however, both control and data messages follow exactly the path detailed in Section 3.2.3. Because the cache–network system's CPU immediately processes all messages by default, control and data messages need not be differentiated.

To better isolate the effects of different message sizes, we used two sizes of messages in our experiments—128-word "long" (data) and 4-word "short" (control) messages—and varied their relative volume while maintaining a constant message rate. Figures 4.30–4.36 graph the results for each application, and Figures 4.37–4.38 graph the combined results of the memory–network and cache–network systems, respectively.

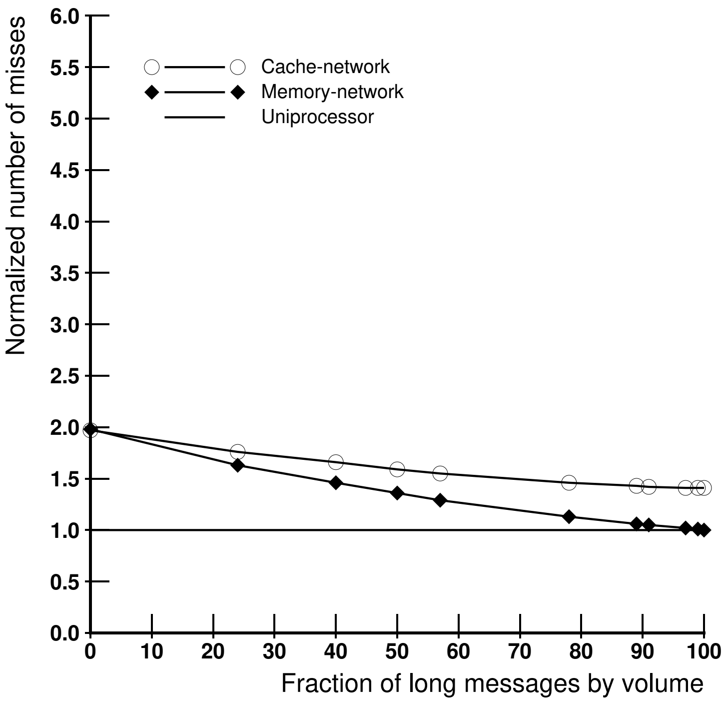


Figure 4.30: Normalized number of misses vs. fraction of long messages by volume (`pthor`)

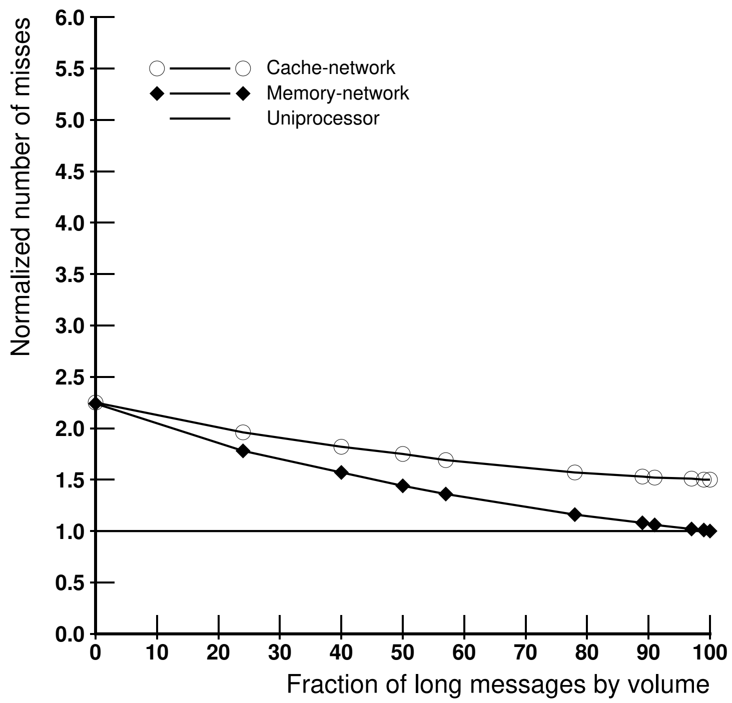


Figure 4.31: Normalized number of misses vs. fraction of long messages by volume (`doduc`)

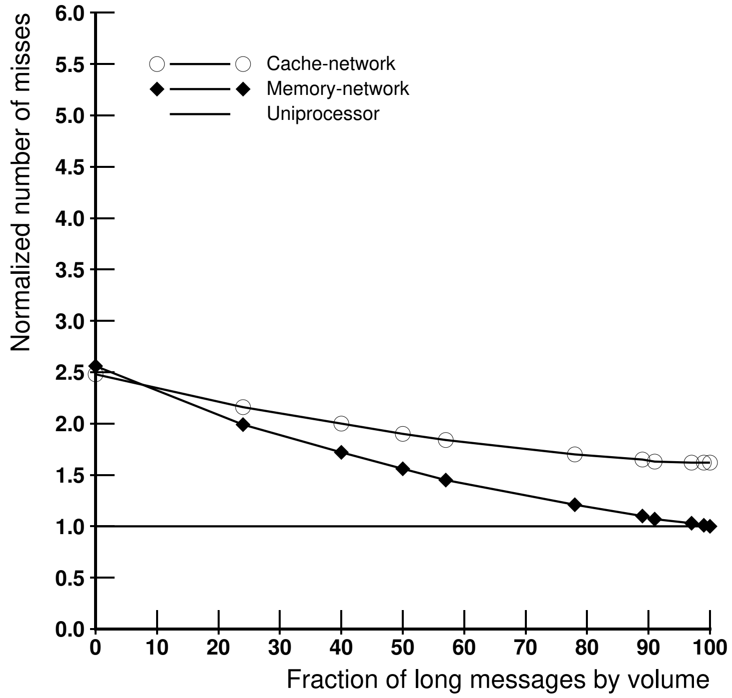


Figure 4.32: Normalized number of misses vs. fraction of long messages by volume (`mdljsp2`)

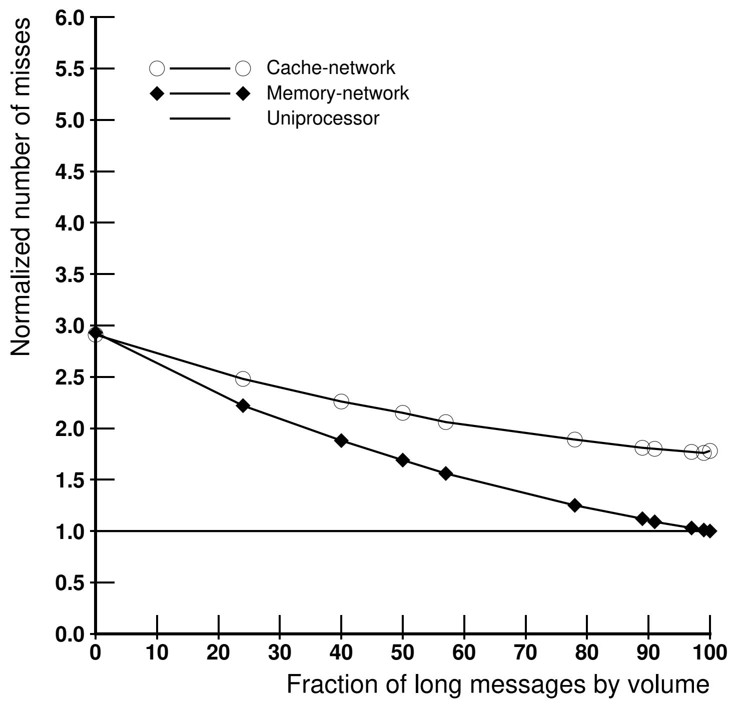


Figure 4.33: Normalized number of misses vs. fraction of long messages by volume (`xlisp`)

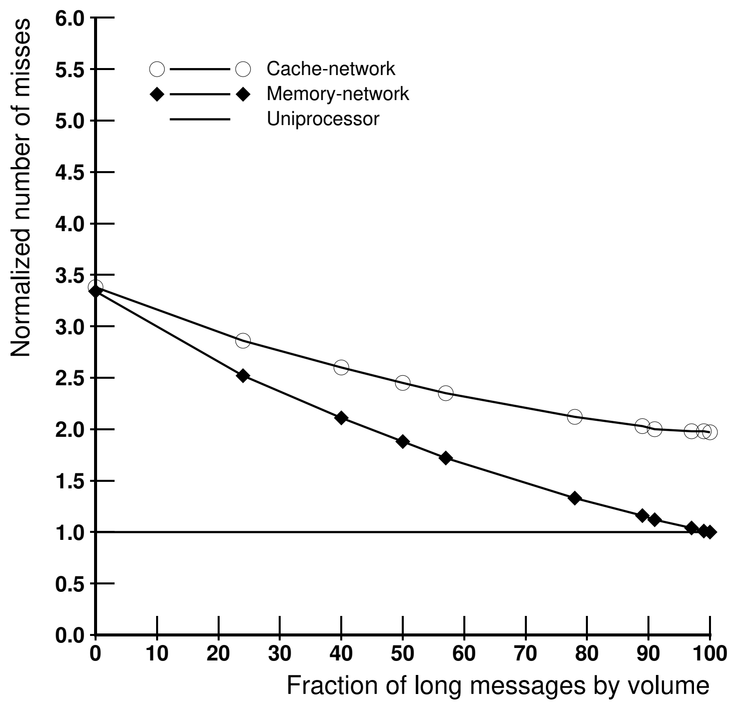


Figure 4.34: Normalized number of misses vs. fraction of long messages by volume (`barnes`)

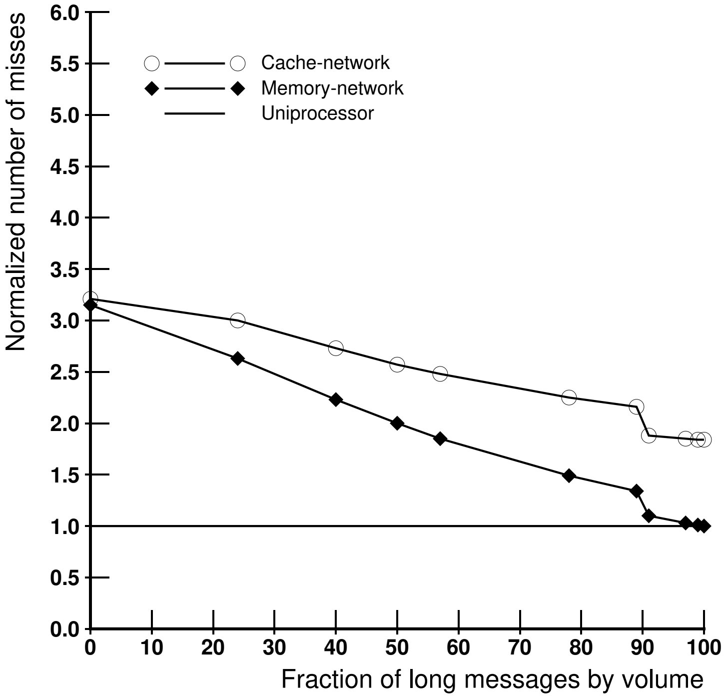


Figure 4.35: Normalized number of misses vs. fraction of long messages by volume (`fpppp`)

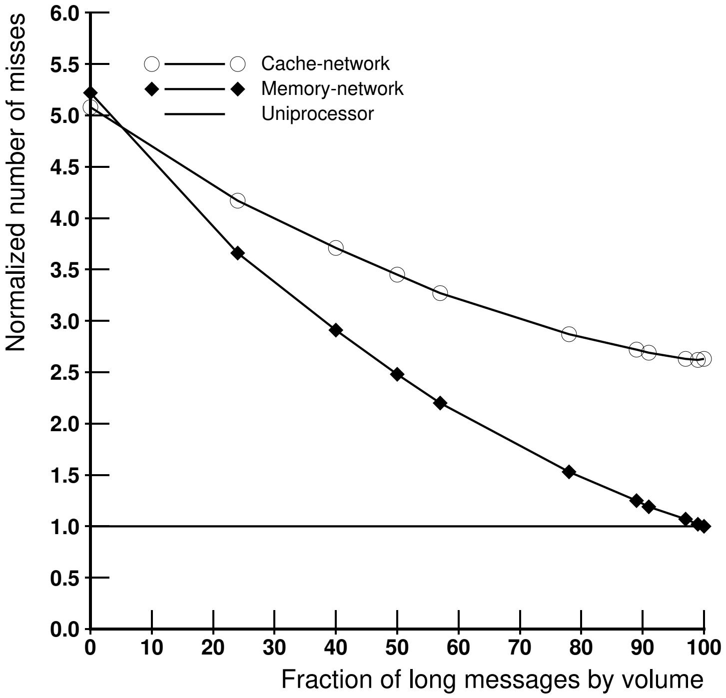


Figure 4.36: Normalized number of misses vs. fraction of long messages by volume (`ear`)

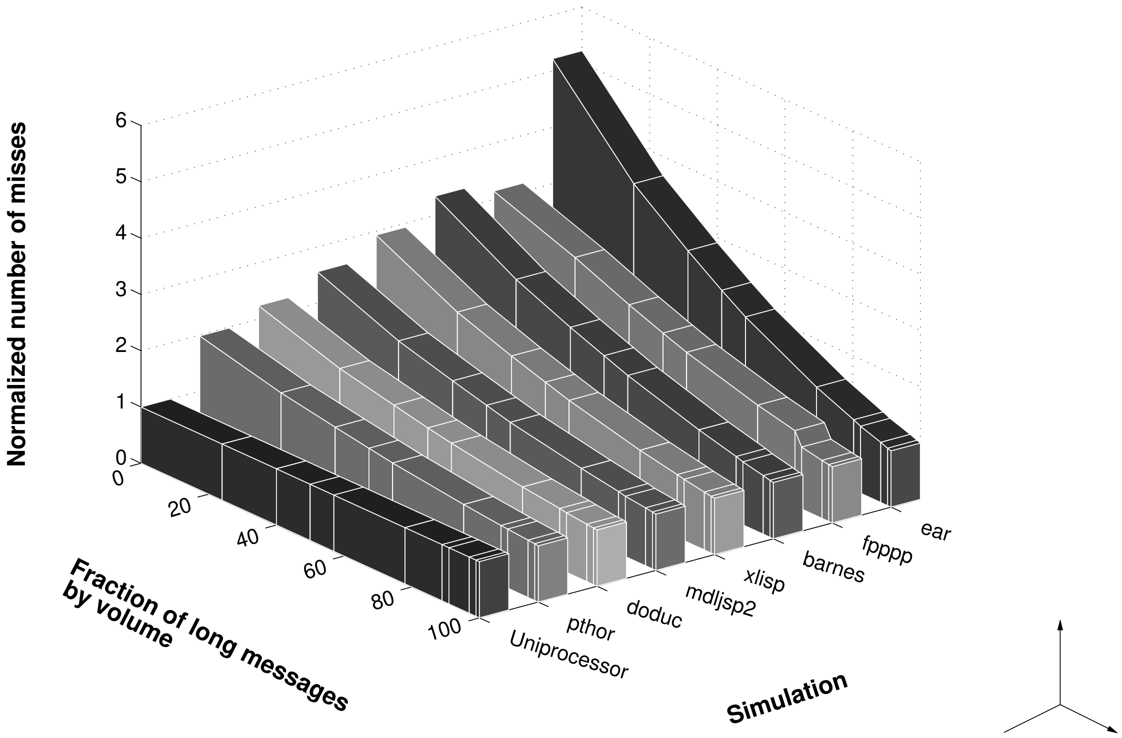


Figure 4.37: Normalized misses vs. fraction of long messages by volume (memory–network)

The insight granted by Figures 4.37–4.38 is that short messages increase cache misses more than long messages on both the cache–network and primary memory–network systems. As the long message fraction is varied from 0%–100%, the number of misses decreases an average of 38% on the cache–network system and an average of 64% on the primary memory–network system. The much higher number of cache misses for short messages is due the lack of temporal and spatial locality inherent in control messages. Control messages are read into the cache, where they displace program data and are then used only briefly.

Observe that the cache–network and memory–network curves representing some of the applications (`pthor`, `mdljsp2`, `xlisp`, and `ear`), intersect (Figures 4.30, 4.32, 4.33, and 4.36). For the rest of the applications, the cache–network curve is higher than the memory–network curve at all points. From this, we gain a second insight: in terms of cache performance, the cache–network system is *sometimes* superior to the primary memory–network system at handling short messages, while the primary memory–network system is *always* superior to the cache–network system at handling long messages. The reason long messages influence the cache less on the primary memory–network system than on the cache–network system is that in the former, message data does not pass through the cache, while in the latter, every message transfer displaces cached data. For short messages, however, the cache–network system has an advantage over the primary memory–network system for message reception. Although both systems read short messages into CPU registers, the cache–network system reads directly from network interface

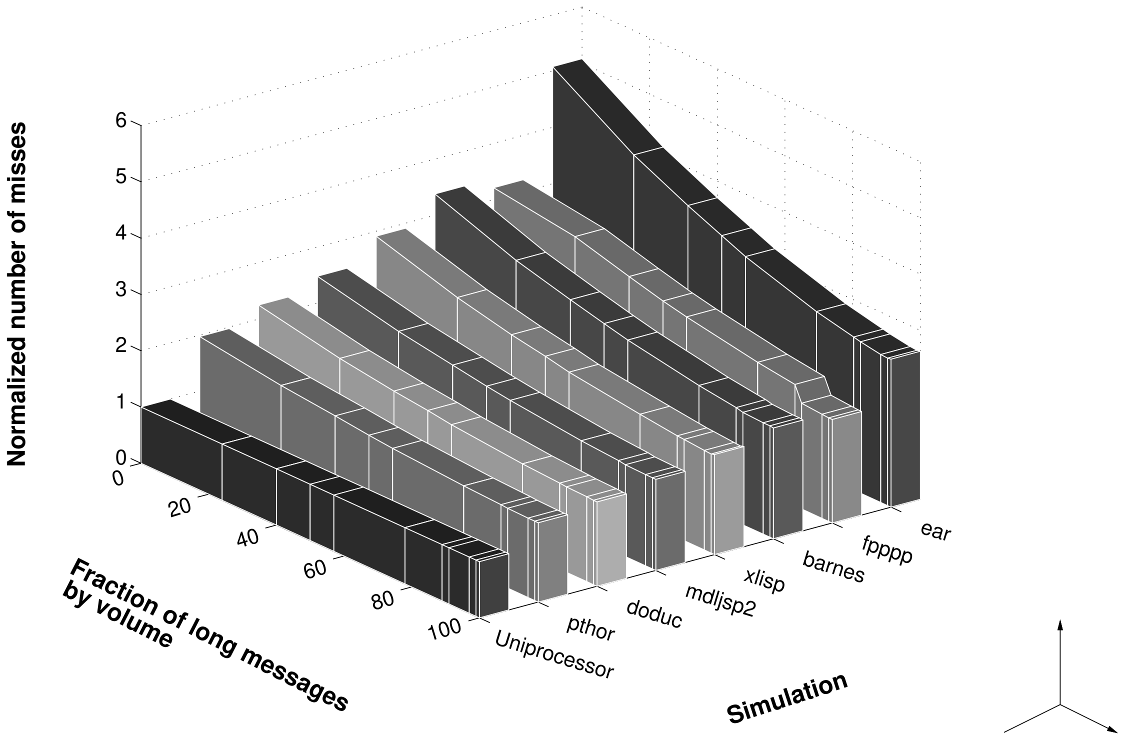


Figure 4.38: Normalized misses vs. fraction of long messages by volume (cache–network)

registers, while the primary memory–network system reads from primary memory. Resultingly, reading short messages causes cache displacement only in the primary memory–network system, not in the cache–network system. Sending short messages, however, causes cache misses in the cache–network system, but not in the primary memory–network system.

The reason some applications favor the cache–network system and others favor the primary memory–network system when sending short messages is as follows. In all applications, the cache–network system both sends and receives a larger volume of messages than the primary memory–network system because the message rate is fixed (at 0.0200 words/clock), but the cache–network system runs for a longer period of time (due to the larger number of cache misses). However, the ratio of cache–network to primary memory–network send volume and the ratio of cache–network to primary memory–network receive volume vary across applications. Applications that use exclusively short messages favor the cache–network system over the primary memory–network system if the expression:

$$\frac{\text{cache–network send volume}}{\text{primary memory–network send volume}} - \frac{\text{cache–network receive volume}}{\text{primary memory–network receive volume}}$$

is less than some threshold value. Figure 4.39 plots the value of the above expression for each application. (Note that, to emphasize the differences between data points, the y-axis is not zero-origined.) As indicated by Figure 4.39, the expression's threshold value is approximately 0.222;

applications corresponding to lesser values favor the cache–network system for short messages, while applications corresponding to greater values favor the primary memory–network system. Furthermore, the distance of each bar from the threshold roughly corresponds to the distance between the lines in Figures 4.30–4.36 at the 0% long messages position.

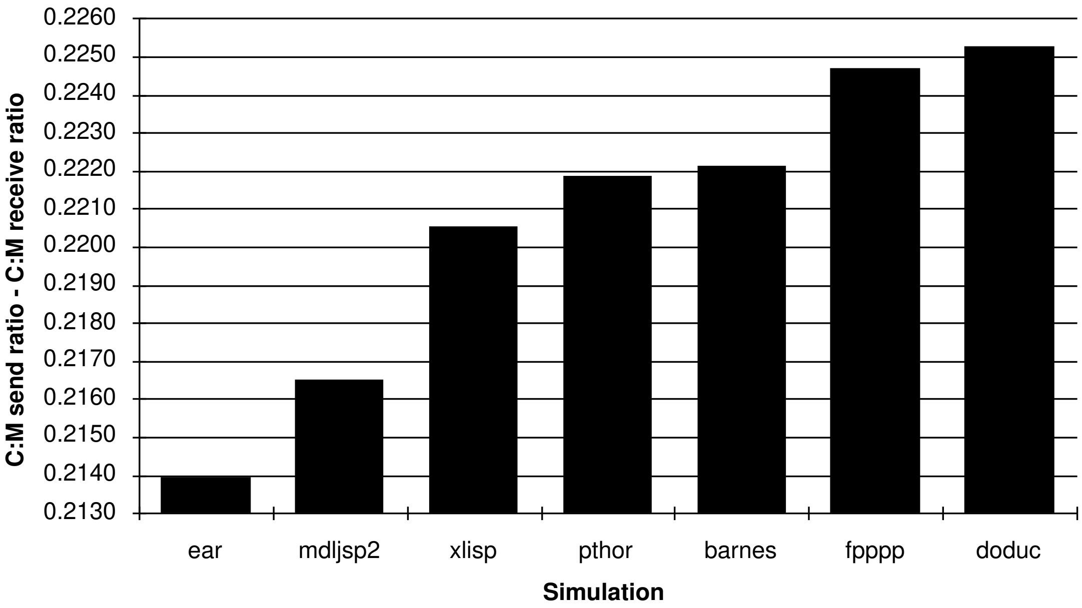


Figure 4.39: Difference of multicomputer send and receive ratios (sorted)

Chapter 5

Discussion

While the previous chapter presented a quantitative analysis of applications' sensitivity to message handling in a variety of scenarios, this chapter examines messaging sensitivity in a more qualitative manner. First, we interpret the previously-presented data and describe the knowledge gained from our results (Section 5.1). Then, in Section 5.2, we draw upon current trends in multicomputer design to forecast trends in multicomputer memory hierarchy performance.

5.1 Present Effects of Message Handling on the Cache

Current mainstream multicomputer systems logically connect the network to either the cache or primary memory. The former scheme—used, for example, in the CM-5—is optimized for low-latency short messages, while the latter—used, for example, in the Paragon—is optimized for high-bandwidth long messages. Table 5.1 confirms that claim using data obtained from [17] and [35]. (The latter reference cites the performance of a Paragon running SUNMOS/PUMA, an operating system and software architecture geared towards more efficient messaging than is provided with the software bundled with the Paragon.) Clock frequencies listed are for the processor clock.

System	Clock freq. (MHz)	Messaging latency (μs)	Bandwidth (MB/s)	Messaging latency (clocks)	Bandwidth (bytes/clock)
CM-5	32	3 – 7	4 – 15	96 – 224	0.125 – 0.469
Paragon	50	30 – 50	19 – 167	1,500 – 2,500	0.380 – 3.340

Table 5.1: Latency and bandwidth of the CM-5 and Paragon

While different network interfaces exhibit different levels of latency and bandwidth, there are other factors involved in determining parallel computer performance. In this thesis, we proposed that cache pollution produced by handling messages is also an important consideration.

The results obtained in Section 4.2 confirm our hypothesis that message handling impacts cache performance. As Figures 4.2–4.9 show, message handling pollutes the cache slightly when running `pthor`, `doduc`, `mdljsp2`, `xlisp`, `barnes`, `fpppp`, and `ear` on a node equipped with a primary memory–network network interface and significantly when running those same applications on a node equipped with a cache–network network interface. It therefore cannot be assumed that the cache miss rate exhibited by a parallel implementation of an application will approximately equal that exhibited by a sequential implementation of the same application, even with equal amounts of application-related work per node. This is an important consideration when writing parallel programs, because it implies that the more a program communicates, the more parallelism is required to maintain a given level of performance. Even the primary memory–network network interface is not immune to cache effects; for example, in heavy traffic, `xlisp`, observes over twice as many cache misses on a primary memory–network node as on a uniprocessor (Figure 4.5).

We showed that optimizing for messaging latency by logically connecting the network interface to the cache causes the side effect of additional cache misses—significantly more than the number observed on a system that logically connects the network interface to primary memory. This bears the unfortunate consequence that the tradeoff between latency and bandwidth is not "fair," in the sense that optimizing a network interface for low-latency messaging implies a cache performance penalty, while optimizing for high-bandwidth messaging does not. Of course, if the intended use of the system is for fine-grained applications, a high-latency architecture is inappropriate. Towards the end of Chapter 7, we describe some alternative architectures that may alleviate the impact of message handling on the memory hierarchy.

Short messages are sensitive to messaging latency, because it accounts for a large fraction of the total transmission time. Short messages are, however, comparatively insensitive to bandwidth; the ability to send more words at once means little when there are only a few words to send. In contrast, long messages are more tolerant of messaging latency because it accounts for a comparatively small fraction of the total transmission time—overhead is amortized over a much larger number of words. But long messages do benefit more from higher bandwidth than do short messages because spooling time dominates for long messages. The greater the rate at which a message can be transmitted, the sooner it will arrive.

In Section 4.4, we showed that short and long messages exhibit different effects on the cache. Because of their poorer spatial locality, short messages cause more cache pollution than long messages. For some applications (`pthor`, `mdljsp2`, and `ear`), the cache–network system observed less pollution on short messages than the primary memory–network system. In those applications, the cache–network system reaps an expected double benefit over the primary memory–network system. Not only do short messages benefit from the cache–network system's lower-latency messaging, but they sometimes additionally cause less cache pollution than that which could be expected on the primary memory–network system. On the other hand, the inverse is true for long messages. Figures 4.30–4.38 show that for all seven applications, the primary memory–network system's cache—unlike the cache–network system's—is virtually immune to long messages. Applications that rely on long messages reap a double benefit on the primary memory–network system because they benefit both from the its higher-bandwidth messaging as well as decreased cache pollution relative to the cache–network system.

5.2 Future Effects of Message Handling on the Cache

Extrapolating from our findings in Section 4.3, we conclude that message handling will degrade the performance of future systems more than current systems. According to Figures 4.19–4.26, cache pollution increases with cache size. That conclusion has serious consequences for multicomputer nodes, which tend to contain larger caches with each generation. Primarily, it shows that cache performance might not scale with the performance of other system components.

The implication is that, all other factors being equal, the use of larger caches in multicomputer microprocessors will decrease parallel speedup. This does not mean that large caches are detrimental to performance. On the contrary, large caches typically improve performance, but they cannot be expected to improve performance as much for microprocessors operating in a multicomputer environment as in a standalone environment.

We further hypothesize that in the future, multicomputer applications will communicate more than current multicomputer applications. This hypothesis is based on two factors: First, if fine-grained parallel programming languages such as Id [1], Concurrent Aggregates [6], and Multilisp [20] gain more acceptance, there will be an increased reliance on communication. And second, as massively-parallel computers become commonplace, each mode has at least the potential for increased communication (i.e. there are more nodes with which to communicate). But, as Figures 4.2–4.9 indicate, an increase in message traffic leads to an increase in cache misses. Increases in cache misses lead to decreases in sequential node performance and hence further limit scalability.

Chapter 6

Summary and Conclusions

Parallel computing is still in its infancy, and thus, there are still a number of open questions about the "right" way to architect parallel machines. Of major concern is the network interface, which serves as the gateway between the CPU and the rest of the system. Sequential performance of a parallel machine, as in a uniprocessor, is a function of CPU and memory hierarchy performance. Parallel performance is largely determined by network interface performance and furthermore, how it interacts with the memory hierarchy.

In Chapter 2, we surveyed a number of existing node architectures, with particular emphasis on the level at which each interacts with the memory hierarchy and unique performance optimizations. We placed each node architecture into one of two categories: those that logically connect the network interface to the cache and those that logically connect the network interface to primary memory. The former, discussed in Section 2.1, supports low-latency communication at the cost of additional CPU involvement. The latter, discussed in Section 2.2, frees up the CPU by offloading message transfer to the DMA controller, but increases communication latency. To address the drawbacks of each design, multicomputer nodes have been built to support features such as:

- Network connections to both the cache and primary memory,
- DMA transfers from the cache,
- Network interface execution of common fine-grained operations, and
- Higher levels of integration.

In addition to CPU-involvement and latency issues, another—occasionally overlooked—feature that distinguishes cache–network and primary memory–network interfaces is each one's impact on the cache. Cache performance is vital to node—and therefore, system—performance because cache memory is significantly faster than primary memory. The higher the cache miss rate is, the more the CPU will have to wait for primary memory to supply it with data. It is therefore imperative that the network interface minimize cache disruptions. Clearly, architectures that logically connect the network interface to the cache will observe more cache misses than architectures that logically connect the network interface to primary memory. What was not previously known, however, is the extent of cache pollution caused by message traffic observed by each architecture. In this thesis, we demonstrated that logically connecting the network interface to the cache significantly increases the expected number of cache misses, while logically connecting the network interface to primary memory increases the expected number of cache misses only slightly.

To reach that conclusion, we constructed trace-based register-transfer-level simulators of three machines that are identical except that one connects the network and the cache, one connects the network and primary memory, and the third is a uniprocessor and therefore has no network component (Chapter 3). With the three simulators, we performed a number of experiments to determine the effect interprocessor communication has on the cache in a variety of scenarios (Chapter 4). By varying message rates, cache sizes, and message lengths, we discovered that processing messages significantly impacts cache on both the cache–network and primary memory–network systems. As a result, sequential performance degrades noticeably in the presence of message traffic. While the primary memory–network system observes up to four times as many cache misses as the uniprocessor under heavy message loads, the cache–network system observes up to 17 times the uniprocessor amount. The large increase is primarily because the cache is involved in message handling and, hence, is frequently polluted.

Control messages (short messages with little locality) cause greater cache pollution—and hence, a greater degradation in system performance—than data messages (long messages with much locality). However, our simulated cache–network system outperforms the corresponding primary memory–network system when handling short control messages on certain applications (`pthor`, `mdljsp2`, and `ear`), although the simulated primary memory–network system is always superior at handling long data messages.

The impact of message traffic on the cache is likely to increase in the future. Our results show that larger caches widen the gap between a node's sequential and parallel performance. Although increasing the cache size improves performance for both the uniprocessor and multicomputer nodes we simulated, the performance gain is smaller when a node must additionally handle interprocessor communication. For example, `fpppp` running on a primary memory–network system with an 8KB cache and handling an average of one word of message data every 50 clocks, 25% long messages by volume, observes fewer than 7% more misses than a uniprocessor with an 8KB cache. But when both systems are equipped with a 64KB cache, the primary memory–network system observes over 31% more misses than the uniprocessor.

The implications of our study are that one cannot assume that a microprocessor will perform equally well on message-passing applications as on sequential applications. Furthermore, there is a drawback to cache-based message handling. While it provides for low-latency communication, sending and receiving messages through the cache results in significant cache pollution. The increased number of cache misses is definitely an important, yet underestimated, factor contributing to poor speedups of parallel programs over their sequential counterparts.

Chapter 7

Future Work

7.1 More Accurate Simulators

Tradeoffs between simulation accuracy and real-time performance must be made in any simulation-based research project. While we intentionally chose to normalize implementation details (for reasons expressed previously), there are still some details that could have been added to the simulators to improve their accuracy, at the expense of increased wall clock time. Note, however, that wall clock time is a serious concern in simulation-based research; the data used in this thesis took over 1.4 SPARCstation[1]-*years* to generate—and that excludes false starts and such real-world occurrences as power outages, machine crashes, and unscheduled system maintenance work (and the corresponding computer downtime).

Simulator accuracy could be enhanced by including support for non-memory instructions, especially if executed superscalarly, as in the DECchip 21064-AA and most modern microprocessors. Doing so would additionally provide simulated execution time figures, something unavailable from memory-only simulations.

Another way to improve simulator accuracy is to simulate multiple nodes instead of a single node surrounded by a network fringe. That way, sender-receiver patterns could be more accurately modelled. That is, a simulator could model blocking receives, in which a node does not perform any additional computation until it receives a message from some source. By simulating additional nodes, simulators behavior will more closely model multicomputer behavior, which can potentially provide additional insights our simulators are incapable of.

7.2 Larger Traces

Although the traces we used range in size from 58–294 million memory references, that represents only about 4–21 seconds of time spent in the memory hierarchy on a system that averages a 70 ns per memory reference (a conservative figure in the context of caches). In contrast, the applications studied in [11] range in duration from 180–17,520 seconds.[2] Note, however, that Cypher, et. al.'s study is not simulation-based; their measurements are taken from monitors on actual multicomputers and were therefore gathered in real time. Traces of more "realistic" (i.e. long-running and computationally-intensive) scientific applications would help researchers

[1] Primarily SPARCstation 10s and 2s

[2] Including time spent outside of the memory hierarchy

estimate the computation, communication, and memory demands of large applications, such as those on the scale of Grand Challenge problems [10]. It would also be beneficial to trace parallel applications (including both memory and communication) to increase the simulations' correspondence to reality, although doing so comes at the expense of control over simulation parameters.

In addition, traces could be augmented with operating system or run-time system references. We did simulate the behavior of a minimal run-time system by generating memory references to transport data between the network interface and memory. However, real messaging software may have additional work to perform "behind the scenes," such as managing virtual memory, updating tables and data structures used for context switching, and potentially explicitly flushing the cache. It would therefore be useful to know whether messaging software increases or decreases the memory hierarchy's sensitivity to message handling.

7.3 Additional Experiments

We simulated only direct-mapped level-one data caches. While direct-mapped caches are the most common, additional insight could be gained from studying set-associative caches (caches in which a data word can be placed in one of multiple cache locations), which might be more resilient to some of the thrashing caused by message handling. For example, `ear` suffers whenever its small (about 9 KB) set of often-used data is displaced from a the cache. It is possible that a two-way set-associative cache would be capable of storing both communication data and computation data simultaneously, thereby reducing the cache miss rate.

Another avenue for future research is to simulate level-two caches. While few current multicomputers are equipped with a level-two cache, it is likely that level-two caches will become more common in the future because they reduce the frequency primary memory latencies are incurred. It is conceivable that the presence of a level-two cache will cause a cache–network system to incur only negligibly more primary memory latencies than a correspondingly-equipped uniprocessor system. In that case, cache–network systems would be able to sustain greater message rates and be usable for a greater range of applications.

7.4 Alternative Architectures

The basic cache–network and primary memory–network designs do not represent the only ways to interface a processor's memory hierarchy and its network communication system. A worthy subject of future research might examine the effects of message passing on other architectures, especially those that connect the network interface and other levels of the memory hierarchy or multiple levels of the memory hierarchy.

We have already begun one such effort by starting to examine the relationship of network traffic to cache misses in the proposed DI-multicomputer [7, 8], in which messages can be routed either directly from registers or memory, and the programmer or compiler chooses which is more appropriate on a per-message basis. Because of the additional flexibility in the DI-multicomputer scheme over the cache–network and primary memory–network architectures, one would expect that machine to perform quite well relative to the architectures discussed in this thesis.

A more evolutionary architecture worth comparing to the cache–network and primary memory–network architectures is one that combines the two machines' network interfaces, letting the programmer decide on a per-message basis whether to route messages between the network and cache or the network and primary memory. We would expect such an architecture, like the DI-multicomputer, to perform well relative to the single-logical-connection architectures evaluated in this thesis.

Finally, a variety of architectures were described in Sections 2.1–2.2. A comprehensive comparison would be interesting, and could provide invaluable information about how each variation from the "plain vanilla" cache–network and primary memory–network designs alters the impact of message handling on the local memory hierarchy.

Appendix A

Pseudocode for Simulator Modules

A.1 CPU

A.1.1 Cache–Network Simulator

```
repeat
    ▷ Move a message from network to memory hierarchy upon arrival
    if an interrupt occured then
        let
            L_recv ← message length (input from the network interface)
            B ← base memory address at which to store the message
        in
            for i ← 0 to L_recv − 1 do
                Input a message word, W, from a memory-mapped network interface register
                Output W to memory location B + 8i
            end
        end
    else
        ▷ Periodically send a message from the memory hierarchy to the network
        if T_send = T_now then
            let
                0 ≤ R_1, R_2 < 1 (randomly)
                L_send ← { L_long   if R_1 < P_long
                         { L_short  if R_1 ≥ 1 − P_long
                B ← base memory address from which to read the message
```

```
            T_send ← T_now − log(2×R_2×Lavg / message rate) network clocks[1]
        in
            Output destination processor number to network interface
            for i ← 0 to L_send − 1 do
                Input a word, W from memory location B + 8i
                Output W to the network interface
            end
        end
    else
        ▷ Ordinary memory operations
        if the trace data queue ≠ ∅ then
            Dequeue an [address | read/write] pair
            Output to memory the address and read/write flag              if read
                             the address, read/write flag, and data       if write
        else
            Output a memory barrier (see [13])
            done ← TRUE
        end
    end
end
while not done
```

A.1.2 Primary Memory–Network Simulator

```
repeat
    ▷ Tell DMA to move a message from network to memory upon arrival
    if an interrupt occured then
        let
            L_recv ← message length (input from the DMA controller)
            B ← base memory address at which to store the message
        in
            Output [address=B | read/write=write | length=L_recv] to the DMA controller's reg-
            isters
            if L_recv ≤ L_short then
                for i ← 0 to L_recv − 1 do
                    Input a message word from memory location B + i
                end
            end
        end
    else
        ▷ Periodically tell DMA to send a message from memory to the network
        if T_send = T_now then
```

[1]The message rate is divided by eight because there are four channels that must together produce $\frac{\text{message rate}}{2}$ words per clock. The CPU produces an additional $\frac{\text{message rate}}{2}$ words per clock, for the desired total.

let

$0 \leq R_1, R_2 < 1$ (randomly)

$L_{send} \leftarrow \begin{cases} L_{long} & \textbf{if } R_1 < P_{long} \\ L_{short} & \textbf{if } R_1 \geq 1 - P_{long} \end{cases}$

$B \leftarrow$ base memory address from which to read the message

$T_{send} \leftarrow T_{now} - \log(\frac{2 \times R_2 \times L_{avg}}{\text{message rate}})$ network clocks

$D \leftarrow$ destination node

in

Output

address=B	read/write=*read*	length=L_{recv}	destination=*dest*

to the DMA controller's registers

end

else

▷ Ordinary memory operations

if the trace data queue $\neq \emptyset$ **then**

Dequeue an $\boxed{\text{address}}\boxed{\text{read/write}}$ pair

Output to memory the address and read/write flag **if** read

the address, read/write flag, and data **if** write

else

Output a memory barrier (see [13])

done $\leftarrow$ TRUE

end

end

end

while not *done*

A.2 Cache

when a (physical) address, A, read/write flag, F, and data word, D arrive from the CPU **do**

if A is cacheable **then**

Split the 31-bit A into a $\overbrace{\boxed{tag}}^{21}\overbrace{\boxed{index}}^{8}\overbrace{\boxed{offset}}^{2}$ triple

if $C[index]_{valid}$ = TRUE **and** $C[index]_{tag} = tag$ **then**

▷ Cache hit

if F=READ **then**

Output $C[index]_{data[offset]}$ to the CPU

else

$C[index]_{data[offset]} \leftarrow D$

end

else

▷ Cache miss

if F=READ **then**

Output FLUSH to the write buffer

Output $\boxed{\text{read}}\boxed{8 \times \lfloor \frac{A}{8} \rfloor}$ to the bus interface

Input cache line, $\mathbf{D} = [D_0, D_1, D_2, D_3]$ from the bus interface

$C[index]_{data[0\ldots3]} \leftarrow \mathbf{D}$
Output $C[index]_{data[offset]}$ to the CPU
end
end
else
▷ A is not cacheable
if F=READ **then**
Output $\boxed{\text{read}}\boxed{8 \times \left\lfloor \frac{A}{8} \right\rfloor}$ to the bus interface
Input data word, D from the bus interface
Output D to the CPU
end
end
end

A.3 Write Buffer

$T_{write} \leftarrow T_{write} + 1$
when a (physical) address, A, read/write flag, F, and data word, D, arrive from the CPU **and** F = WRITE **do**
if A is cacheable **then**
Split the 31-bit A into a $\overbrace{\boxed{tag}}^{29}\overbrace{\boxed{offset}}^{2}$ pair
if $\exists i, 0 \leq i < 4$ such that $W[i]_{tag} = tag$ **then**
▷ Line is already in write buffer
$W[i]_{data[offset]} \leftarrow D$
$W[i]_{mask[offset]} \leftarrow$ VALID
else
▷ Line is not already in write buffer
if W is full **then**
Output $W[head]_{data[0\ldots3]}$ and $W[head]_{mask[0\ldots3]}$ to the bus interface starting at address $\boxed{W[head]_{tag}}\boxed{00}$
Input DONE from the bus interface
Dequeue $W[head]$
end
Enqueue $W[tail]$ **with**
$W[tail]_{tag} = tag$
$W[tail]_{mask[0\ldots3 \text{ (except } offset)]} =$ INVALID
$W[tail]_{mask[offset]} =$ VALID
$W[tail]_{data[offset]} = D$
end
end
$T_{write} \leftarrow T_{now}$
else
▷ A is not cacheable
Output D and A to the bus interface
Input DONE from the bus interface

end
end
if ($W_{validentries} > 2$) **or** ($W_{validentries} > 0$ **and** $T_{now} - T_{write} \geq 256$) **then**
Output $W[head]_{data}[0\ldots3]$ and $W[head]_{mask}[0\ldots3]$ to the bus interface starting at address $\boxed{W[head]_{tag}}\boxed{00}$
Input DONE from the bus interface
Dequeue $W[head]$
else
if we just input a "flush" directive **then**
for each i **from** *head* **to** *tail* **do**
Output $W[i]_{data}[0\ldots3]$ and $W[i]_{mask}[0\ldots3]$ to the bus interface starting at address $\boxed{W[i]_{tag}}\boxed{00}$
Input DONE from the bus interface
Dequeue $W[i]$
end
end
end

A.4 Bus Interface

when a (physical) address, A, read/write flag, F, and (**if** F=WRITE) data words, **D** arrive from the cache or write buffer **do**
wait until bus is not occupied
case

Had bus last	Cache wants bus	Write buffer wants bus	**of**
—	False	True	⟶ Write buffer gets the bus
—	True	False	⟶ Cache gets the bus
Cache	True	True	⟶ Write buffer gets the bus
Write buffer	True	True	⟶ Cache gets the bus

end
if write buffer has the bus **then**
Output A, F, and $\mathbf{D}[0,1]$ to the bus
Output A, F, and $\mathbf{D}[2,3]$ to the bus
Output DONE to the write buffer
else
Output A and F to the bus
Output DONE to the cache
Input $\mathbf{D}'[0,1]$ from the bus
Input $\mathbf{D}'[2,3]$ from the bus
Output $\mathbf{D}'$ to the cache
end
end

A.5 Bus

repeat forever
    Input a (physical) address, A, read/write flag, F, data words, $\mathbf{D}$, and mask, $\mathbf{M}$
    Output A, F, $\mathbf{D}$, and $\mathbf{M}$ to the bus interface, memory controller, and network interface (cache–network) or DMA controller (primary memory–network)
end

A.6 Memory and Memory Controller

repeat forever
    for each $i \in \{0, 2\}$ **do**
        Input a (physical) address, A, read/write flag, F, data words, $\mathbf{D}[i, i+1]$, and mask, $\mathbf{M}$ from the bus
        if F = READ **then**
            Output $\mathbf{D}[i, i+1]$ to the bus
        end
    end
end

A.7 Network Interface

A.7.1 Cache–Network Simulator

when data is on the bus **do**
    Input a (physical) address, A, read/write flag, F, and data words, $D[0, 1]$ from the bus
    if $A = A_{reg}$ **then**
        if F = WRITE **then**
            ▷ Send data from the bus to the router
            $dest \leftarrow D[0]$
            $L \leftarrow D[1]$
            Input data words $D[2, 3]$ from the bus and ignore
            Output $dest$ to the router
            Output L to the router
            for $i \leftarrow 0$ **to** $L - 1$ **do**
                Input data words, $D[i \bmod 2, i \bmod 2 + 1]$ from the bus
                Output $D[i \bmod 2]$ to the router
                Output $D[i \bmod 2 + 1]$ to the router
            end
            Output DONE to the router
        else
            ▷ Send data from the message FIFO to the bus
            for $i \leftarrow 0$ **to** 3 **do**
                wait until $R \neq \emptyset$
                Dequeue R into $D[i]$
            end

```
            Output D[0,1] to the bus
            Output D[2,3] to the bus
        end
    end
end
when data is at the router do
    ▷ Buffer data read from the router
    Output an interrupt to the CPU
    while data is at the router do
        Input data word D' from the router
        Enqueue D' onto queue R
        if R is full then
            Output WAIT to the router
            wait until R is no longer full
        end
        Output CONTINUE to the router
    end
end
```

A.7.2 Primary Memory–Network Simulator

```
when data is on the bus do
    Input a (physical) address, A, read/write flag, F, and data words, D[0,1] from the bus
    Input data words D[2,3] from the bus
    if A = A_reg and F = WRITE then
        let
            A_mem ← D[0]
            op ← D[1]
            dest ← D[2]
            L ← D[3]
        in
            if op = SEND then
              ▷ Send data from memory to the router
              Output bus request to bus interface
              Input HAVE-BUS from bus interface
              Output dest to the router
              Output L to the router
              for i ← 0 to L − 1 by 32 do
                  Output | READ | A_mem + i | to memory via the bus
                  Input data words D[0,1] from memory via the bus
                  Input data words D[2,3] from memory via the bus
                  Output D[0], D[1], D[2], and D[3] to the router
              end
              Output DONE to the router
              Output DONE to the bus interface
            else
```

```
                ▷ Send data to memory from the router
                Output bus request to bus interface
                Input HAVE-BUS from bus interface
                Dequeue dest from R
                Dequeue L from R
                for i ← 0 to L − 1 by 32 do
                    Dequeue R four times—into D[0], D[1], D[2], and D[3], respectively
                    Output | WRITE | A_mem + i      | D[0] | D[1] | to memory via the bus
                    Output | WRITE | A_mem + i + 16 | D[2] | D[3] | to memory via the bus
                end
                Output DONE to the bus interface
            end
        end
    end
end
when data is at the router do
    ▷ Buffer data read from the router
    Output an interrupt to the CPU
    while data is at the router do
        Input data word D′ from the router
        Enqueue D′ onto queue R
        if R is full then
            Output WAIT to the router
            wait until R is no longer full
        end
        Output CONTINUE to the router
    end
end
```

A.8 Network Fringe

```
repeat forever
    ▷ Send data on each of four channels at a random interval
    for i ← 0 to 3 do
        if T_send[i] = T_now then
            let
                0 ≤ R_1, R_2 < 1 (randomly)
                L_send ← { L_long   if R_1 < P_long
                         { L_short  if R_1 ≥ 1 − P_long
                B ← base memory address from which to read the message
                T_send[i] ← T_now − log((8×R_2×L_avg)/(message rate)) network clocks
            in
                Output L_send to the network interface on channel i
```

```
            for j ← 0 to L_send − 1 do
              Input a word, W from memory location B + 8j
              Output W to the network interface on channel i
            end
          end
      end
    end
end
```

Appendix B

Derivation of "Reasonable" Message Rates

We used the measurements of computer and application characteristics shown in Table B.1 to derive the average message rate shown in Table B.2.

Application ([11], p. 3)	Computer ([11], p. 3)	MHz	Procs. ([11], p. 3)	Msg. vol. (GB) ([11], p. 7)	Time (sec.) ([11], p. 10)
`CLIMATE`	Delta	40	256	965	292
`SEMI`	Delta	40	512	120	108
`MOLECULE`	nCUBE/2	20	512	1,956	59
`RENDER`	Delta	40	32	2	3
`EXFLOW`	Delta	40	512	562	216
`QCD`	nCUBE/1	10	256	7	133
`VORTEX`	nCUBE/2	20	64	1	24
`REACT`	Delta	40	512	132	132

Table B.1: Computer and application characteristics

Application	$\frac{\text{64-bit words}}{\text{processor}}$	$\frac{\text{Time (clocks)}}{1{,}000{,}000}$	$\frac{\text{64-bit words}}{\text{clock}}$
`CLIMATE`	471,191,406	700,800	0.00067
`SEMI`	29,296,875	259,200	0.00011
`MOLECULE`	477,539,063	70,800	0.00674
`RENDER`	7,812,500	7,200	0.00109
`EXFLOW`	137,207,031	518,400	0.00026
`QCD`	3,417,969	79,800	0.00004
`VORTEX`	1,953,125	28,800	0.00007
`REACT`	32,226,563	316,800	0.00010
			$\mu = 0.00114$

Table B.2: Derivation of average message rate

Appendix C

Raw Data

This appendix lists the raw data obtained from all our simulations. Each table lists the number of cache misses and the number of messages and words transmitted for the cache–network and primary memory–network architectures, and the number of cache missses for the uniprocessor architecture.[1] Note that there is a slight amount of repetition in the following tables; simulations run with all the default parameters (0.0200 words/clock, 16 KB cache size, and 91% long messages by volume—see Section 3.4) are listed in each of the following tables.

C.1 Varying Message Rate

The tables in this section vary the message rate while maintaining a 16 KB cache and 91% long messages by volume.

Words/ clock	**Cache–network**			**Memory–network**			**Uniproc.**
	Misses	Messages	Words	Misses	Messages	Words	Misses
0.0010	5,277,727	14,278	516,583	5,206,865	14,176	513,132	5,195,877
0.0020	5,361,482	28,969	1,027,016	5,217,692	28,777	1,060,296	5,195,877
0.0100	6,135,884	170,594	6,060,066	5,312,693	153,392	5,649,624	5,195,877
0.0200	7,394,673	413,228	14,675,768	5,450,503	333,868	12,403,916	5,195,877
0.0400	10,837,204	1,220,918	43,228,514	5,828,998	812,764	30,036,512	5,195,877
0.0600	15,440,609	2,767,314	97,746,596	6,432,947	1,528,849	56,542,172	5,195,877
0.0800	20,692,565	5,512,272	194,290,390	7,308,250	2,481,796	91,343,036	5,195,877
0.1000	26,921,532	11,447,674	402,852,468	8,649,772	3,826,785	140,606,156	5,195,877

Table C.1: `pthor`, varying message rate, 16 KB cache, 91% long messages by volume

[1] No messages are transmitted on a uniprocessor.

Words/ clock	Cache–network			Memory–network			Uniproc.
	Misses	Messages	Words	Misses	Messages	Words	Misses
0.0010	6,980,048	20,282	725,426	6,869,979	20,412	755,804	6,852,588
0.0020	7,108,607	41,535	1,473,680	6,887,541	40,907	1,510,836	6,852,588
0.0100	8,347,365	246,330	8,690,802	7,039,936	219,728	8,112,876	6,852,588
0.0200	10,426,947	609,595	21,652,225	7,261,239	480,551	17,725,796	6,852,588
0.0400	16,303,502	1,874,289	66,452,746	7,869,867	1,171,310	43,308,380	6,852,588
0.0600	24,615,755	4,428,839	156,410,790	8,845,832	2,216,504	81,769,028	6,852,588
0.0800	34,543,218	9,169,884	323,470,011	10,275,260	3,643,627	134,393,652	6,852,588
0.1000	46,728,204	19,692,971	692,848,446	12,504,399	5,704,001	209,589,192	6,852,588

Table C.2: `doduc`, varying message rate, 16 KB cache, 91% long messages by volume

Words/ clock	Cache–network			Memory–network			Uniproc.
	Misses	Messages	Words	Misses	Messages	Words	Misses
0.0010	4,550,833	17,253	601,393	4,464,340	16,875	622,628	4,450,743
0.0020	4,660,966	35,178	1,256,237	4,479,002	34,488	1,250,896	4,450,743
0.0100	5,663,867	206,320	7,332,772	4,600,620	184,905	6,839,572	4,450,743
0.0200	7,276,824	505,832	17,911,414	4,781,222	406,397	15,062,300	4,450,743
0.0400	11,838,720	1,518,393	53,972,703	5,281,040	994,286	36,767,980	4,450,743
0.0600	17,979,560	3,490,131	123,163,104	6,087,975	1,891,624	69,925,420	4,450,743
0.0800	24,903,404	7,026,837	247,826,110	7,275,107	3,110,038	114,622,132	4,450,743
0.1000	33,035,541	14,641,284	514,929,758	9,101,687	4,864,350	178,755,808	4,450,743

Table C.3: `mdljsp2`, varying message rate, 16 KB cache, 91% long messages by volume

Words/ clock	Cache–network			Memory–network			Uniproc.
	Misses	Messages	Words	Misses	Messages	Words	Misses
0.0010	2,631,733	11,471	401,488	2,570,077	11,197	416,424	2,560,334
0.0020	2,713,927	23,252	827,612	2,580,484	22,979	841,524	2,560,334
0.0100	3,424,910	137,426	4,905,793	2,666,382	121,479	4,484,016	2,560,334
0.0200	4,606,304	342,294	12,158,827	2,795,696	268,920	9,983,700	2,560,334
0.0400	7,906,971	1,056,179	37,420,847	3,149,070	660,356	24,432,388	2,560,334
0.0600	12,308,896	2,452,622	86,666,760	3,719,531	1,257,849	46,378,344	2,560,334
0.0800	17,199,944	4,942,006	174,311,444	4,569,149	2,089,863	76,905,916	2,560,334
0.1000	22,832,646	10,277,642	361,453,026	5,860,773	3,286,657	120,770,780	2,560,334

Table C.4: `xlisp`, varying message rate, 16 KB cache, 91% long messages by volume

Words/ clock	**Cache–network**			**Memory–network**			**Uniproc.**
	Misses	Messages	Words	Misses	Messages	Words	Misses
0.0010	2,554,380	13,650	482,130	2,470,822	13,342	487,344	2,457,834
0.0020	2,645,822	27,613	961,175	2,483,609	26,709	977,188	2,457,834
0.0100	3,538,678	163,730	5,823,011	2,596,668	145,035	5,365,016	2,457,834
0.0200	4,925,018	406,687	14,351,126	2,760,534	318,439	11,828,044	2,457,834
0.0400	8,711,077	1,246,339	44,203,298	3,198,713	780,831	28,880,772	2,457,834
0.0600	13,691,265	2,877,444	101,670,580	3,893,881	1,493,350	55,088,516	2,457,834
0.0800	19,293,980	5,780,082	203,894,514	4,897,881	2,477,914	91,344,496	2,457,834
0.1000	25,930,741	12,060,836	424,023,695	6,421,405	3,919,451	144,216,544	2,457,834

Table C.5: `barnes`, varying message rate, 16 KB cache, 91% long messages by volume

Words/ clock	**Cache–network**			**Memory–network**			**Uniproc.**
	Misses	Messages	Words	Misses	Messages	Words	Misses
0.0010	7,898,855	32,545	1,146,458	7,688,751	31,672	1,169,992	7,656,101
0.0020	8,143,789	66,239	2,346,354	7,721,020	64,405	2,367,800	7,656,101
0.0100	10,469,683	396,566	14,085,485	8,004,818	346,210	12,761,872	7,656,101
0.0200	14,419,481	1,009,474	35,856,197	8,420,061	762,037	28,262,396	7,656,101
0.0400	28,929,244	3,514,732	124,466,952	11,215,570	2,021,176	74,655,596	7,656,101
0.0600	47,378,730	8,720,010	308,314,379	13,177,856	3,869,537	142,896,640	7,656,101
0.0800	69,152,816	18,600,177	656,070,586	16,093,202	6,458,486	238,044,792	7,656,101
0.1000	96,078,740	40,766,386	1,433,399,823	20,731,724	10,347,006	380,366,972	7,656,101

Table C.6: `fpppp`, varying message rate, 16 KB cache, 91% long messages by volume

Words/ clock	**Cache–network**			**Memory–network**			**Uniproc.**
	Misses	Messages	Words	Misses	Messages	Words	Misses
0.0010	4,010,471	29,594	1,062,326	3,813,446	29,152	1,085,312	3,783,517
0.0020	4,244,923	60,506	2,158,915	3,844,390	58,473	2,157,156	3,783,517
0.0100	6,458,681	363,397	12,920,144	4,109,601	315,663	11,677,040	3,783,517
0.0200	10,185,551	932,185	33,099,587	4,506,569	699,747	25,863,184	3,783,517
0.0400	20,535,185	3,006,412	106,469,685	5,610,036	1,745,341	64,509,460	3,783,517
0.0600	33,724,678	7,106,078	250,846,871	7,432,353	3,401,910	125,713,408	3,783,517
0.0800	47,605,124	14,273,612	503,468,570	10,174,337	5,786,899	213,283,352	3,783,517
0.1000	63,441,773	29,578,815	1,040,620,422	14,441,939	9,402,593	345,719,088	3,783,517

Table C.7: `ear`, varying message rate, 16 KB cache, 91% long messages by volume

C.2 Varying Cache Size

The tables in this section vary the cache size while maintaining a message rate of 0.0200 words/clock cache and 91% long messages by volume.

Cache	**Cache–network**			**Memory–network**			**Uniproc.**
size (KB)	Misses	Messages	Words	Misses	Messages	Words	Misses
8	9,082,532	476,556	16,868,530	6,799,216	383,252	14,156,268	6,499,826
16	7,394,673	413,228	14,675,768	5,450,503	333,868	12,403,916	5,195,877
32	6,195,803	368,368	13,113,251	4,458,663	297,459	10,929,424	4,227,456
64	5,085,675	325,819	11,556,635	3,514,181	262,302	9,681,144	3,305,886

Table C.8: `pthor`, 0.0200 words/clock, varying cache, 91% long messages by volume

Cache	**Cache–network**			**Memory–network**			**Uniproc.**
size (KB)	Misses	Messages	Words	Misses	Messages	Words	Misses
8	14,824,134	772,391	27,458,671	10,672,528	604,915	22,375,360	10,142,228
16	10,426,947	609,595	21,652,225	7,261,239	480,551	17,725,796	6,852,588
32	7,669,033	504,739	17,839,767	5,099,508	399,805	14,728,424	4,754,300
64	6,192,851	451,171	16,033,680	3,811,286	351,411	12,988,944	3,496,619

Table C.9: `doduc`, 0.0200 words/clock, varying cache, 91% long messages by volume

Cache	**Cache–network**			**Memory–network**			**Uniproc.**
size (KB)	Misses	Messages	Words	Misses	Messages	Words	Misses
8	13,368,701	719,405	25,564,869	9,883,850	580,782	21,461,260	9,418,728
16	7,276,824	505,832	17,911,414	4,781,222	406,397	15,062,300	4,450,743
32	6,281,973	467,312	16,653,019	4,014,843	378,673	14,028,756	3,709,819
64	4,040,185	385,765	13,679,091	2,113,068	311,406	11,578,464	1,853,952

Table C.10: `mdljsp2`, 0.0200 words/clock, varying cache, 91% long messages by volume

Cache size (KB)	Cache–network			Memory–network			Uniproc.
	Misses	Messages	Words	Misses	Messages	Words	Misses
8	6,080,464	398,334	14,156,922	3,957,730	312,545	11,596,880	3,679,856
16	4,606,304	342,294	12,158,827	2,795,696	268,920	9,983,700	2,560,334
32	3,827,812	312,114	11,055,007	2,202,676	245,890	9,090,692	1,977,926
64	2,865,940	275,064	9,732,240	1,322,849	212,163	7,875,196	1,106,114

Table C.11: `xlisp`, 0.0200 words/clock, varying cache, 91% long messages by volume

Cache size (KB)	Cache–network			Memory–network			Uniproc.
	Misses	Messages	Words	Misses	Messages	Words	Misses
8	7,296,512	509,033	18,088,014	4,763,232	404,369	14,982,312	4,419,615
16	4,925,018	406,687	14,351,126	2,760,534	318,439	11,828,044	2,457,834
32	3,742,295	359,196	12,734,507	1,738,721	275,732	10,187,652	1,468,130
64	2,741,493	318,190	11,260,133	1,036,587	246,767	9,128,076	805,680

Table C.12: `barnes`, 0.0200 words/clock, varying cache, 91% long messages by volume

Cache size (KB)	Cache–network			Memory–network			Uniproc.
	Misses	Messages	Words	Misses	Messages	Words	Misses
8	29,210,020	1,558,912	55,359,722	19,761,836	1,179,064	43,453,712	18,539,704
16	14,419,481	1,009,474	35,856,197	8,420,061	762,037	28,262,396	7,656,101
32	6,818,687	709,908	25,250,295	2,786,415	538,451	19,914,628	2,265,442
64	5,350,232	650,868	23,066,686	1,962,612	506,604	18,787,004	1,497,548

Table C.13: `fpppp`, 0.0200 words/clock, varying cache, 91% long messages by volume

Cache size (KB)	Cache–network			Memory–network			Uniproc.
	Misses	Messages	Words	Misses	Messages	Words	Misses
8	17,830,576	1,235,075	43,782,691	11,124,773	956,031	35,328,128	10,213,495
16	10,185,551	932,185	33,099,587	4,506,569	699,747	25,863,184	3,783,517
32	6,625,324	790,091	28,023,626	2,251,418	611,206	22,683,980	1,698,138
64	5,577,861	747,645	26,456,267	1,925,226	598,182	22,138,432	1,457,226

Table C.14: `ear`, 0.0200 words/clock, varying cache, 91% long messages by volume

C.3 Varying Message Size

The tables in this section vary the fraction of long messages by volume while maintaining a message rate of 0.0200 words/clock and a 16 KB cache.

Long msg. vol. (%)	**Cache–network**			**Memory–network**			**Uniproc.**
	Misses	Messages	Words	Misses	Messages	Words	Misses
0	10,231,694	4,763,388	21,311,761	10,310,737	4,642,420	27,512,304	5,195,877
24	9,140,323	3,253,623	18,590,716	8,473,221	3,057,490	22,026,188	5,195,877
40	8,614,014	2,483,845	17,296,299	7,585,067	2,279,276	19,216,740	5,195,877
50	8,286,679	2,013,163	16,481,071	7,068,291	1,816,032	17,596,948	5,195,877
57	8,078,674	1,699,240	16,038,404	6,723,849	1,509,452	16,529,804	5,195,877
78	7,584,259	885,492	14,934,170	5,895,833	753,854	13,888,876	5,195,877
89	7,424,681	501,100	14,633,336	5,532,131	410,449	12,621,616	5,195,877
91	7,394,673	413,228	14,675,768	5,450,503	333,868	12,403,916	5,195,877
97	7,330,220	219,478	14,619,557	5,284,210	174,498	11,811,416	5,195,877
99	7,318,069	150,756	14,718,999	5,225,496	116,623	11,570,344	5,195,877
100	7,304,176	114,676	14,735,107	5,196,030	88,894	11,556,112	5,195,877

Table C.15: `pthor`, 0.0200 words/clock, 16 KB cache, varying volume of long messages

Long msg. vol. (%)	**Cache–network**			**Memory–network**			**Uniproc.**
	Misses	Messages	Words	Misses	Messages	Words	Misses
0	15,432,561	7,377,040	33,018,241	15,336,790	7,052,366	41,848,200	6,852,588
24	13,462,589	4,950,183	28,307,586	12,210,763	4,555,597	32,777,212	6,852,588
40	12,504,773	3,741,001	25,937,882	10,738,028	3,359,982	28,406,920	6,852,588
50	11,974,923	3,018,012	24,751,875	9,879,121	2,663,179	25,777,548	6,852,588
57	11,600,676	2,536,793	23,926,131	9,314,392	2,198,962	24,003,700	6,852,588
78	10,791,300	1,315,262	22,205,160	7,975,207	1,086,616	19,908,052	6,852,588
89	10,476,936	739,080	21,550,415	7,388,476	589,806	18,129,060	6,852,588
91	10,426,947	609,595	21,652,225	7,261,239	480,551	17,725,796	6,852,588
97	10,317,114	325,056	21,617,883	6,994,022	248,938	16,914,084	6,852,588
99	10,282,096	221,092	21,587,753	6,900,470	167,834	16,572,928	6,852,588
100	10,264,212	168,068	21,595,654	6,852,799	126,301	16,419,888	6,852,588

Table C.16: `doduc`, 0.0200 words/clock, 16 KB cache, varying volume of long messages

Long msg. vol. (%)	Cache–network			Memory–network			Uniproc.
	Misses	Messages	Words	Misses	Messages	Words	Misses
0	11,041,232	5,911,714	26,447,704	11,415,222	5,976,878	35,398,616	4,450,743
24	9,595,638	4,015,407	22,958,499	8,852,831	3,868,595	27,837,340	4,450,743
40	8,882,633	3,050,757	21,177,949	7,640,588	2,854,684	24,057,108	4,450,743
50	8,470,810	2,470,100	20,223,763	6,929,198	2,258,165	21,886,236	4,450,743
57	8,189,341	2,079,777	19,587,687	6,471,893	1,869,563	20,515,780	4,450,743
78	7,560,228	1,083,088	18,238,031	5,363,280	919,374	16,923,696	4,450,743
89	7,337,827	612,764	17,979,821	4,886,290	498,198	15,327,212	4,450,743
91	7,276,824	505,832	17,911,414	4,781,222	406,397	15,062,300	4,450,743
97	7,202,338	268,959	17,869,704	4,564,956	209,599	14,234,304	4,450,743
99	7,193,235	184,421	17,980,870	4,489,174	141,140	13,991,312	4,450,743
100	7,193,930	139,898	17,976,447	4,450,862	107,139	13,930,224	4,450,743

Table C.17: `mdljsp2`, 0.0200 words/clock, 16 KB cache, varying volume of long messages

Long msg. vol. (%)	Cache–network			Memory–network			Uniproc.
	Misses	Messages	Words	Misses	Messages	Words	Misses
0	7,443,590	4,181,617	18,714,713	7,498,929	4,076,456	24,185,460	2,560,334
24	6,341,111	2,803,316	15,995,760	5,688,630	2,615,033	18,808,040	2,560,334
40	5,796,351	2,116,715	14,691,827	4,824,344	1,920,054	16,230,788	2,560,334
50	5,500,106	1,710,258	14,019,157	4,323,595	1,513,370	14,673,696	2,560,334
57	5,283,268	1,430,406	13,493,253	3,998,387	1,252,754	13,703,004	2,560,334
78	4,827,823	741,605	12,530,731	3,211,096	612,983	11,285,952	2,560,334
89	4,639,969	415,631	12,137,794	2,871,180	331,111	10,197,304	2,560,334
91	4,606,304	342,294	12,158,827	2,795,696	268,920	9,983,700	2,560,334
97	4,540,987	182,403	12,094,197	2,640,369	138,647	9,410,956	2,560,334
99	4,510,218	123,662	12,068,589	2,587,511	93,705	9,298,068	2,560,334
100	4,545,758	95,091	12,219,000	2,560,419	70,228	9,130,024	2,560,334

Table C.18: `xlisp`, 0.0200 words/clock, 16 KB cache, varying volume of long messages

Long msg. vol. (%)	Cache–network			Memory–network			Uniproc.
	Misses	Messages	Words	Misses	Messages	Words	Misses
0	8,298,789	5,035,992	22,541,301	8,199,725	4,780,526	28,390,768	2,457,834
24	7,022,641	3,376,191	19,300,804	6,181,785	3,091,046	22,259,996	2,457,834
40	6,391,566	2,545,019	17,716,903	5,196,151	2,277,249	19,243,856	2,457,834
50	6,011,248	2,051,935	16,828,595	4,610,726	1,798,195	17,474,572	2,457,834
57	5,764,990	1,716,960	16,153,939	4,222,684	1,484,445	16,228,628	2,457,834
78	5,218,416	884,243	14,940,655	3,274,518	726,510	13,380,860	2,457,834
89	4,986,681	495,735	14,493,497	2,852,373	390,720	12,020,704	2,457,834
91	4,925,018	406,687	14,351,126	2,760,534	318,439	11,828,044	2,457,834
97	4,875,373	217,480	14,439,733	2,562,903	164,069	11,157,736	2,457,834
99	4,862,247	148,631	14,470,961	2,492,513	110,096	10,907,128	2,457,834
100	4,843,564	112,549	14,461,733	2,457,919	82,701	10,752,180	2,457,834

Table C.19: `barnes`, 0.0200 words/clock, 16 KB cache, varying volume of long messages

Long msg. vol. (%)	Cache–network			Memory–network			Uniproc.
	Misses	Messages	Words	Misses	Messages	Words	Misses
0	24,589,816	12,950,434	57,968,838	24,141,111	12,216,177	72,545,904	7,656,101
24	22,977,597	9,159,906	52,369,860	20,133,479	8,245,236	59,343,180	7,656,101
40	20,898,473	6,850,231	47,530,242	17,070,512	5,999,937	50,711,680	7,656,101
50	19,706,812	5,485,845	44,930,778	15,307,082	4,713,374	45,717,488	7,656,101
57	18,959,891	4,589,867	43,274,474	14,157,956	3,876,153	42,349,108	7,656,101
78	17,195,363	2,349,723	39,684,310	11,432,659	1,882,157	34,629,076	7,656,101
89	16,529,329	1,319,626	38,597,298	10,250,721	1,010,806	31,182,952	7,656,101
91	14,419,481	1,009,474	35,856,197	8,420,061	762,037	28,262,396	7,656,101
97	14,167,811	533,441	35,486,661	7,919,747	392,081	26,656,116	7,656,101
99	14,121,310	364,945	35,575,379	7,745,052	262,427	25,959,536	7,656,101
100	14,067,787	276,124	35,480,189	7,656,717	199,327	25,914,432	7,656,101

Table C.20: `fpppp`, 0.0200 words/clock, 16 KB cache, varying volume of long messages

Long msg. vol. (%)	**Cache–network**			**Memory–network**			**Uniproc.**
	Misses	Messages	Words	Misses	Messages	Words	Misses
0	19,234,075	11,929,684	53,393,731	19,736,246	11,890,001	70,571,704	3,783,517
24	15,775,332	7,902,876	45,105,540	13,847,785	7,424,230	53,449,528	3,783,517
40	14,051,630	5,916,761	41,123,079	11,024,655	5,349,920	45,301,964	3,783,517
50	13,043,499	4,736,730	38,784,683	9,394,410	4,168,195	40,412,976	3,783,517
57	12,369,772	3,961,872	37,367,491	8,329,399	3,404,876	37,186,056	3,783,517
78	10,855,601	2,022,982	34,080,304	5,804,377	1,621,166	29,837,368	3,783,517
89	10,279,045	1,133,635	33,170,791	4,735,922	864,009	26,673,000	3,783,517
91	10,185,551	932,185	33,099,587	4,506,569	699,747	25,863,184	3,783,517
97	9,948,358	493,217	32,795,616	4,031,059	359,513	24,445,132	3,783,517
99	9,923,707	338,020	32,970,773	3,866,478	240,048	23,785,084	3,783,517
100	9,946,199	257,797	33,125,842	3,784,669	181,053	23,537,672	3,783,517

Table C.21: `ear`, 0.0200 words/clock, 16 KB cache, varying volume of long messages

Appendix D

Data to Accompany Graphs

This appendix provides the raw data used to generate the graphs in Chapter 4.

D.1 Data Corresponding to Section 4.2

Words/ clock	**Uniproc.**	**Memory–network**						
	All	pthor	doduc	mdljsp2	xlisp	barnes	fpppp	ear
0.0010	1.0000	1.0021	1.0025	1.0031	1.0038	1.0053	1.0043	1.0079
0.0020	1.0000	1.0042	1.0051	1.0063	1.0079	1.0105	1.0085	1.0161
0.0100	1.0000	1.0225	1.0273	1.0337	1.0414	1.0565	1.0455	1.0862
0.0200	1.0000	1.0490	1.0596	1.0743	1.0919	1.1232	1.0998	1.1911
0.0400	1.0000	1.1219	1.1485	1.1866	1.2299	1.3014	1.4649	1.4828
0.0600	1.0000	1.2381	1.2909	1.3679	1.4528	1.5843	1.7212	1.9644
0.0800	1.0000	1.4065	1.4995	1.6346	1.7846	1.9928	2.1020	2.6891
0.1000	1.0000	1.6647	1.8248	2.0450	2.2891	2.6126	2.7079	3.8171

Table D.1: Normalized number of misses vs. message rate (memory–network)

Words/ clock	Uniproc.	Cache–network						
	All	pthor	doduc	mdljsp2	xlisp	barnes	fpppp	ear
0.0010	1.0000	1.0158	1.0186	1.0225	1.0279	1.0393	1.0317	1.0600
0.0020	1.0000	1.0319	1.0374	1.0472	1.0600	1.0765	1.0637	1.1220
0.0100	1.0000	1.1809	1.2181	1.2726	1.3377	1.4398	1.3675	1.7071
0.0200	1.0000	1.4232	1.5216	1.6350	1.7991	2.0038	1.8834	2.6921
0.0400	1.0000	2.0857	2.3792	2.6599	3.0883	3.5442	3.7786	5.4275
0.0600	1.0000	2.9717	3.5922	4.0397	4.8075	5.5705	6.1884	8.9136
0.0800	1.0000	3.9825	5.0409	5.5953	6.7179	7.8500	9.0324	12.5822
0.1000	1.0000	5.1813	6.8191	7.4225	8.9178	10.5502	12.5493	16.7679

Table D.2: Normalized number of misses vs. message rate (cache–network)

Words/ clock	Uniproc.	Memory–network						
	All	pthor	doduc	mdljsp2	xlisp	barnes	fpppp	ear
0.0010	1.0000	1.002	1.003	1.003	1.004	1.005	1.004	1.008
0.0020	1.0000	1.004	1.005	1.006	1.008	1.011	1.009	1.016
0.0100	1.0000	1.023	1.027	1.034	1.041	1.057	1.046	1.086
0.0200	1.0000	1.049	1.060	1.074	1.092	1.123	1.100	1.191
0.0400	1.0000	1.122	1.149	1.187	1.230	1.301	1.465	1.483
0.0600	1.0000	1.238	1.291	1.368	1.453	1.584	1.721	1.964
0.0800	1.0000	1.407	1.500	1.635	1.785	1.993	2.102	2.689
0.1000	1.0000	1.665	1.825	2.045	2.289	2.613	2.708	3.817

Table D.3: Normalized number of "non-sending" misses vs. message rate (memory–network)

Words/ clock	Uniproc.	Cache–network						
	All	pthor	doduc	mdljsp2	xlisp	barnes	fpppp	ear
0.0010	1.0000	1.003	1.005	1.005	1.008	1.014	1.012	1.024
0.0020	1.0000	1.006	1.010	1.011	1.017	1.027	1.024	1.049
0.0100	1.0000	1.034	1.059	1.065	1.097	1.142	1.136	1.276
0.0200	1.0000	1.077	1.135	1.145	1.218	1.293	1.310	1.621
0.0400	1.0000	1.183	1.330	1.350	1.506	1.600	2.018	2.384
0.0600	1.0000	1.314	1.582	1.595	1.821	1.919	2.633	3.064
0.0800	1.0000	1.458	1.857	1.850	2.116	2.238	3.254	3.602
0.1000	1.0000	1.622	2.165	2.125	2.420	2.599	3.925	4.119

Table D.4: Normalized number of "non-sending" misses vs. message rate (cache–network)

D.2 Data Corresponding to Section 4.3

Cache size (KB)	**Uniproc.**	**Memory–network**						
	All	`pthor`	`doduc`	`mdljsp2`	`xlisp`	`barnes`	`fpppp`	`ear`
8	1.000	1.046	1.052	1.049	1.076	1.078	1.066	1.089
16	1.000	1.049	1.060	1.074	1.092	1.123	1.100	1.191
32	1.000	1.055	1.073	1.082	1.114	1.184	1.230	1.326
64	1.000	1.063	1.090	1.140	1.196	1.287	1.311	1.321

Table D.5: Normalized number of misses vs. cache size (memory–network)

Cache size (KB)	**Uniproc.**	**Cache–network**						
	All	`pthor`	`doduc`	`mdljsp2`	`xlisp`	`barnes`	`fpppp`	`ear`
8	1.000	1.397	1.462	1.419	1.652	1.651	1.576	1.746
16	1.000	1.423	1.522	1.635	1.799	2.004	1.883	2.692
32	1.000	1.466	1.613	1.693	1.935	2.549	3.010	3.902
64	1.000	1.538	1.771	2.179	2.591	3.403	3.573	3.828

Table D.6: Normalized number of misses vs. cache size (cache–network)

Cache size (KB)	**Uniprocessor**	**Memory–network**	**Cache–network**
8	6,499,826	6,799,216	9,082,532
16	5,195,877	5,450,503	7,394,673
32	4,227,456	4,458,663	6,195,803
64	3,305,886	3,514,181	5,085,675

Table D.7: Absolute number of misses vs. cache size (`pthor`)

Cache size (KB)	**Uniprocessor**	**Memory–network**	**Cache–network**
8	10,142,228	10,672,528	14,824,134
16	6,852,588	7,261,239	10,426,947
32	4,754,300	5,099,508	7,669,033
64	3,496,619	3,811,286	6,192,851

Table D.8: Absolute number of misses vs. cache size (`doduc`)

Cache size (KB)	**Uniprocessor**	**Memory–network**	**Cache–network**
8	9,418,728	9,883,850	13,368,701
16	4,450,743	4,781,222	7,276,824
32	3,709,819	4,014,843	6,281,973
64	1,853,952	2,113,068	4,040,185

Table D.9: Absolute number of misses vs. cache size (`mdljsp2`)

Cache size (KB)	**Uniprocessor**	**Memory–network**	**Cache–network**
8	3,679,856	3,957,730	6,080,464
16	2,560,334	2,795,696	4,606,304
32	1,977,926	2,202,676	3,827,812
64	1,106,114	1,322,849	2,865,940

Table D.10: Absolute number of misses vs. cache size (`xlisp`)

Cache size (KB)	**Uniprocessor**	**Memory–network**	**Cache–network**
8	4,419,615	4,763,232	7,296,512
16	2,457,834	2,760,534	4,925,018
32	1,468,130	1,738,721	3,742,295
64	805,680	1,036,587	2,741,493

Table D.11: Absolute number of misses vs. cache size (`barnes`)

Cache size (KB)	**Uniprocessor**	**Memory–network**	**Cache–network**
8	18,539,704	19,761,836	29,210,020
16	7,656,101	8,420,061	14,419,481
32	2,265,442	2,786,415	6,818,687
64	1,497,548	1,962,612	5,350,232

Table D.12: Absolute number of misses vs. cache size (`fpppp`)

Cache size (KB)	**Uniprocessor**	**Memory–network**	**Cache–network**
8	10,213,495	11,124,773	17,830,576
16	3,783,517	4,506,569	10,185,551
32	1,698,138	2,251,418	6,625,324
64	1,457,226	1,925,226	5,577,861

Table D.13: Absolute number of misses vs. cache size (`ear`)

D.3 Data Corresponding to Section 4.4

Fraction long msgs.	Uniproc. All	Memory–network pthor	doduc	mdljsp2	xlisp	barnes	fpppp	ear
0%	1.00	1.98	2.24	2.56	2.93	3.15	3.34	5.22
24%	1.00	1.63	1.78	1.99	2.22	2.63	2.52	3.66
40%	1.00	1.46	1.57	1.72	1.88	2.23	2.11	2.91
50%	1.00	1.36	1.44	1.56	1.69	2.00	1.88	2.48
57%	1.00	1.29	1.36	1.45	1.56	1.85	1.72	2.20
78%	1.00	1.13	1.16	1.21	1.25	1.49	1.33	1.53
89%	1.00	1.06	1.08	1.10	1.12	1.34	1.16	1.25
91%	1.00	1.05	1.06	1.07	1.09	1.10	1.12	1.19
97%	1.00	1.02	1.02	1.03	1.03	1.03	1.04	1.07
99%	1.00	1.01	1.01	1.01	1.01	1.01	1.01	1.02
100%	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table D.14: Normalized number of misses vs. fraction of long messages (memory–network)

Fraction long msgs.	Uniproc. All	Cache–network pthor	doduc	mdljsp2	xlisp	barnes	fpppp	ear
0%	1.00	1.97	2.25	2.48	2.91	3.21	3.38	5.08
24%	1.00	1.76	1.96	2.16	2.48	3.00	2.86	4.17
40%	1.00	1.66	1.82	2.00	2.26	2.73	2.60	3.71
50%	1.00	1.59	1.75	1.90	2.15	2.57	2.45	3.45
57%	1.00	1.55	1.69	1.84	2.06	2.48	2.35	3.27
78%	1.00	1.46	1.57	1.70	1.89	2.25	2.12	2.87
89%	1.00	1.43	1.53	1.65	1.81	2.16	2.03	2.72
91%	1.00	1.42	1.52	1.63	1.80	1.88	2.00	2.69
97%	1.00	1.41	1.51	1.62	1.77	1.85	1.98	2.63
99%	1.00	1.41	1.50	1.62	1.76	1.84	1.98	2.62
100%	1.00	1.41	1.50	1.62	1.78	1.84	1.97	2.63

Table D.15: Normalized number of misses vs. fraction of long messages (cache–network)

Bibliography

[1] Arvind and K. Ekanadham. Future scientific programming on parallel machines. *Journal of Parallel and Distributed Computing*, 5(5):460–493, October 1988.

[2] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. Computer Sciences Technical Report 1031, University of Wisconsin–Madison, 1210 West Dayton St.; Madison, WI 53706, September 1991. Available from `ftp://ftp.cs.wisc.edu/tech-reports/reports/91/tr1031.ps.Z`.

[3] Carl J. Beckmann. CARL: An architecture simulation language. Report 1066, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 305 Talbot, 104 South Wright St.; Urbana, IL 61801-2932, December 1990.

[4] John Bruner. Parsim user interface reference manual. Report 1002, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 305 Talbot, 104 South Wright St.; Urbana, IL 61801-2932, September 1990.

[5] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11):198–206, April 1981.

[6] Andrew A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs*. MIT Press, Cambridge, MA, 1993.

[7] Lynn Choi and Andrew A. Chien. Integrating networks and memory hierarchies in a multicomputer node architecture. In *Proceedings of the International Parallel Processing Symposium*, April 1994. Available from `http://www-csag.cs.uiuc.edu/papers/ipps94.ps`.

[8] Lynn Choi and Andrew A. Chien. The design and performance evaluation of the DI-multicomputer. *Journal of Parallel and Distributed Computing*, 199x. Submitted for publication.

[9] SPEC Steering Committee. `DESCR.n` files in SPEC distribution, January 1992.

[10] Committee on Information and Communication (CIC) of the National Science and Technology Council (NSTC). High performance computing and communications: Technology for the national information infrastructure (Supplement to the president's fiscal year 1995 budget). Available from `http://www.hpcc.gov/blue95`, 1994.

[11] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural requirements of parallel scientific applications with explicit communication. In *Proceedings of the International Symposium on Computer Architecture*, pages 2–13. IEEE Computer Society, 1993.

[12] William J. Dally, J. A. Stuart Fiske, John S. Keen, Richard A. Lethin, Michael D. Noakes, Peter R. Nuth, Roy E. Davison, and Gregory A. Fyler. The message-driven processor. *IEEE Micro*, pages 23–39, April 1992.

[13] Digital Equipment Corporation. *Alpha Architecture Handbook*, 1992. Order number EC-H1689-10.

[14] Digital Equipment Corporation, Maynard, MA. *DECchip 21064-AA Microprocessor Hardware Reference Manual*, 1st edition, October 1992. Order number EC-N0079-72.

[15] Kaivalya M. Dixit. CINT2.0 and CFP2.0 benchmark descriptions. *SPEC Newsletter*, 3(4):18–21, December 1991.

[16] Kaivalya M. Dixit. New CPU benchmark suites from SPEC. In *Thirty-Seventh IEEE Computer Society International Conference*, pages 305–310, Spring 1992.

[17] C. Leiserson et al. The network architecture of the Connection Machine CM-5. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, 1992. Available from `ftp://cmns.think.com/doc/Papers/net.ps.Z`.

[18] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[19] Jeffrey D. Gee, Mark D. Hill, Dionisios N. Pnevmatikatos, and Alan Jay Smith. Cache performance of the SPEC92 benchmark suite. *IEEE Micro*, pages 17–27, August 1993.

[20] Robert H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[21] M. Homewood and M. McLaren. Meiko CS-2 interconnect Elan – Elite design. In *Proceedings of the IEEE Hot Interconnects Symposium.* IEEE TCMM, August 1993.

[22] Intel Corporation. *iPSC/2 and iPSC/860 User's Guide*, June 1990. Order number 311532-006.

[23] Intel Corporation. *Paragon XP/S Product Overview*, 1991.

[24] Hiroaki Ishihata, Takeshi Horie, Satoshi Inano, Toshiyuki Shimizu, and Sadayuki Kato. An architecture of highly parallel computer AP1000. In *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*, pages 13–16, May 1991. Available from `ftp://fcapwide.fujitsu.co.jp/ap1000/english/rim/rim_91.ps.Z`.

[25] John Kubiatowicz and Anant Agarwal. The anatomy of a message in the Alewife multiprocessor. In *Proceedings of the International Conference on Supercomputing*, July 1993. Available from `ftp://cag.lcs.mit.edu/pub/papers/anatomy.ps.Z`.

[26] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. Computer Sciences Technical Report 1083, University of Wisconsin–Madison, 1210 West Dayton St.; Madison, WI 53706, March 1992. Available from `ftp://ftp.cs.wisc.edu/tech-reports/reports/91/tr1083.ps.Z`.

[27] Barry A. Maskas, Stephen F. Shirron, and Nicholas A. Warchol. Design and performance of the DEC 4000 AXP departmental server computing systems. *Digital Technical Journal*, 4(4):1–18, 1992. Available from `ftp://ftp.digital.com/pub/Digital/info/DTJ/axp-dec-4000.ps`.

[28] David May, Roger Shepherd, and Peter Thompson. The T9000 Transputer. Technical report, Inmos Limited, 1000 Aztec West, Almondsbury, Bristol BS12 4SQ, UK, 1993. Available from `ftp://ftp.inmos.co.uk/inmos/info/T9000/T9000.ps.Z`.

[29] John Palmer and Guy L. Steele Jr. Connection machine model CM-5 system overview. In *Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 474–483, 1992.

[30] C. Seitz, N. Boden, J. Seizovic, and W. Su. The design of the Caltech Mosaic C multicomputer. In *Proceedings of the University of Washington Symposium on Integrated Systems*, 1993.

[31] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared memory. Technical report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, Stanford University, CA 94305, April 1991. Available from `ftp://mojave.stanford.edu/pub/splash/report/splash.ps`.

[32] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[33] Craig B. Stunkel and W. Kent Fuchs. An analysis of cache performance for a hypercube multicomputer. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):421–432, July 1992.

[34] Günter Watzlawik and Franz Hutner. A pipelined network interface for a parallel computer. Technical report, Bull/ECRC/ICL/Siemens, ZFE ST SN 22, Siemens AG, Otto-Hahn-Ring 6, 81730 München, Germany, 1993.

[35] Stephen Wheat, Barney Maccabe, Rolf Riesen, David van Dresser, and Mark Stallcup. Overheads from SUNMOS/PUMA presentation, September 1993. Available from `http://www.ccsf.caltech.edu/paragon/sunmos/sunmos_slides.ps`.

[36] Patrick Henry Winston and Berthold Klaus Paul Horn. *LISP*. Addison-Wesley, second edition, 1984.